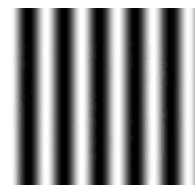
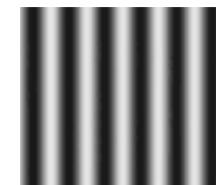


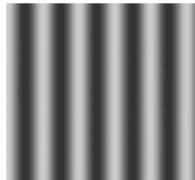
Smallest font



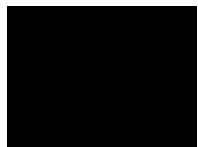
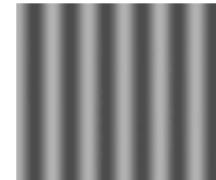
Please turn off and put  
away your cell phone



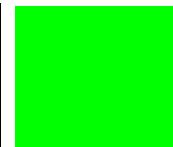
# Calibration slide



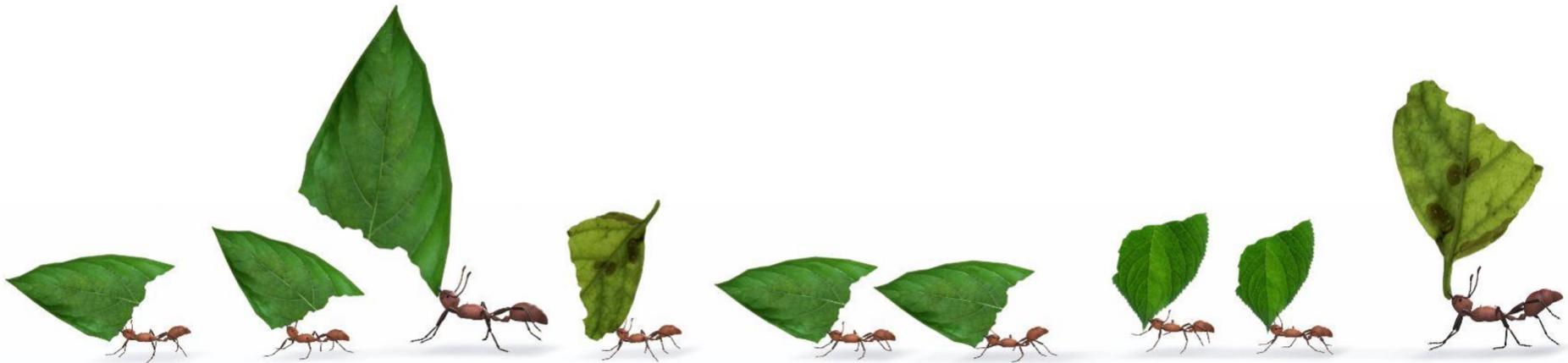
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



For context:  
*Colocasia gigantea*  
(Giant elephant ear)





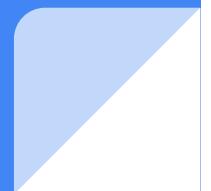
**NYU**

Center for  
Data Science

# Week 01: Welcome to Big Data

## DS-GA 1004: Big Data

Instructor: Pascal Wallisch, PhD



# Lesson plan

- What exactly **\*is\*** Big Data?
- Class logistics
- Philosophy and CS context

What is  
Big Data?

# When I took this class, it seemed to be about cutting-edge tools and frameworks

In today's hyper-scaling data ecosystem, we seamlessly integrate **SQL** queries through **Postgres** alongside unstructured storage in **MongoDB**, while **Hadoop**'s distributed **HDFS** layer—augmented by **Parquet** for efficient columnar storage—and **MapReduce** functions coordinate massive parallel workloads. Simultaneously, **Docker**-based microservices orchestrated by **Kubernetes** simplify container deployment, and **Dask** manages in-memory computations. Meanwhile, **Kafka** handles real-time data pipelines for event-driven processing. Leveraging **Apache Pig** for higher-level scripting and **Spark** for iterative analytics, we adopt a **Dremel**-inspired architecture that unifies multi-source queries and aggregates. Critical data checkpoints rely on **hash functions** to ensure integrity and reliability throughout the pipeline. The entire environment is versioned and managed on **GitHub**, deployed on **AWS** for limitless scalability, and accelerated by **GPU** resources to ensure near-instantaneous insights across all touchpoints.

# What this class is actually about: **Scaling**

In many fields, scale matters (critically):

You can drop a mouse down a thousand-yard  
mine shaft and, on arriving at the bottom, it  
gets a slight shock and walks away.

~1 oz (28.35 g)

A rat is killed.

~1 lb (16 oz)

A man is broken.

~180 lbs (2,880 oz)

A horse splashes.

~1000 lbs (16,000 oz)

--*J.B.S. Haldane, biologist*

In brief:  
Quantity has a  
quality of its own

# What about scaling of data?

Generally speaking, more data, i.e. a larger sample size (more rows) and a richer set of features (more columns) makes everything better in data science:

Specifically:

- Higher power / better recall
- Better parameter estimates
- Better recommendations
- Enables subgroup analysis
- The blessing of dimensionality

...



Can one have too much of a good thing?



# What this looks like in the context of data:



Hey Pascal, question for you. I have a massive massive dataset of genetic data (has over 1000 rows and over 20 million columns!). I need to load it in so that I can clean it and analyze it. But every time I attempt to do so, it uses all 126gigs of my machine's memory to do so, and then it crashes. Any advice?

# Many such cases...

Error using `zeros`

Requested `702x120x1000000` (627.6GB) array exceeds maximum array size preference (64.0GB). This might cause MATLAB to become unresponsive.

A showstopper, if unhandled

Now what?

Operational definition  
big data, *n.*:

Whatever doesn't fit on your laptop...

# More seriously...

- The definition of “big” depends on how the data is used and stored
- In practical terms, “**big data**” is often differentiated by requiring **\*coordinated\* processing by \*multiple\* computers (“machines”).**
- Much of this class will focus on **distributed storage and computation.**

**Note:** We will be using the CS concept of data in this class, not the DS conception.

- DS concept of data: The “givens” (latin). Immutable. Sacred. What we use to gain insights from, do statistics with and use to train machine learning models.
- CS concept of data: Bits and bytes. Digital information that has to be loaded and stored (to/in memory), fed to a processor, changed (by processing it), and sent around.

# Big data is about more than just size: The five “V”s of big data

<b>Volume</b>	The quantity of data
<b>Velocity</b>	Speed at which new data is recorded
<b>Variety</b>	Data may be structured or heterogeneous
<b>Veracity</b>	Data can be noisy, incomplete, or wrong
<b>Value</b>	The benefits and actionable insights within

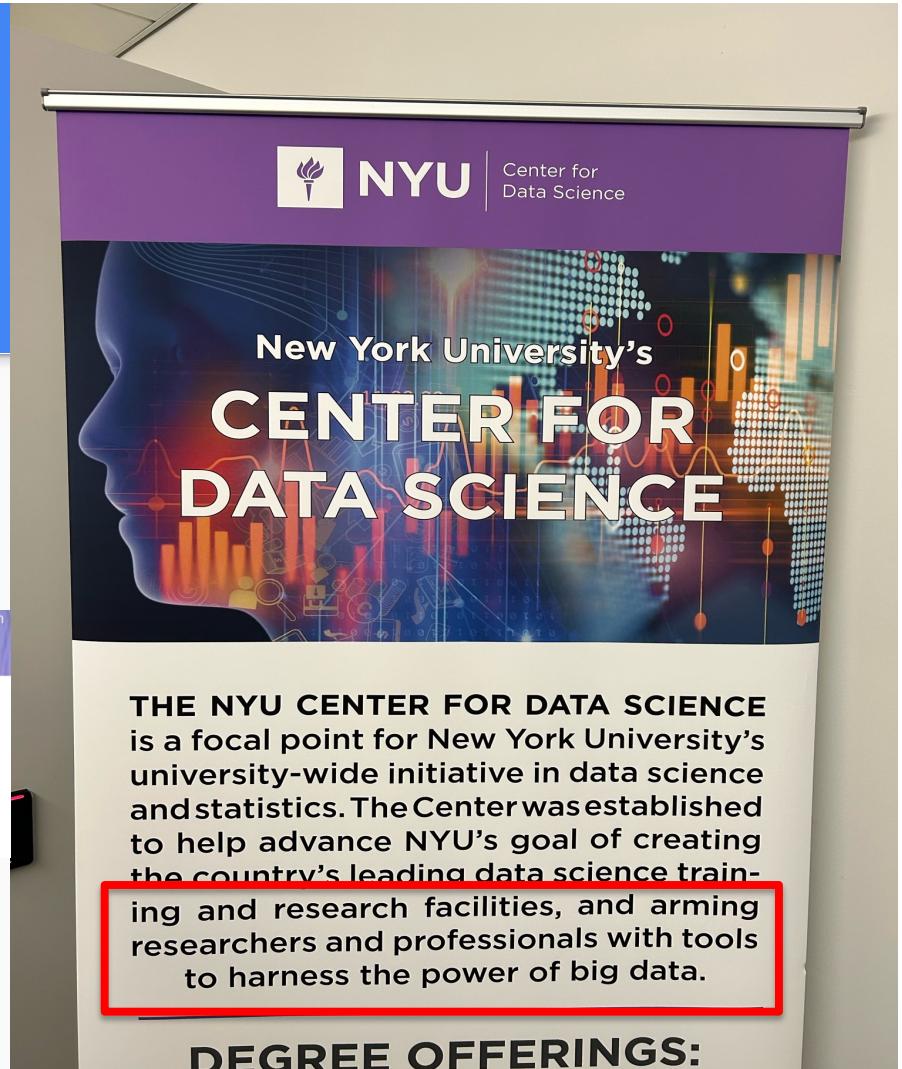
Laney, 2001;  
Hurwitz et al., 2013

# Why is there a whole class on big data?

## Yann's keynote:

Machine Learning sucks! (compared to humans and animals)

- ▶ Supervised learning (SL) requires large numbers of labeled samples.
- ▶ Reinforcement learning (RL) requires insane amounts of trials.
- ▶ Self-Supervised Learning (SSL) works great but...
- ▶ Generative prediction only works for text and other discrete modalities



# So in essence:

- We need data to make machine learning and statistics work well.
- The more the better.
- But: At some point, handling all of this data becomes a challenge in itself
- More data = new problems...
- We'll start discussing later today, which specific problems are introduced by an overwhelming quantity of data

# What did we have in mind when designing this class?

- The tools are constantly evolving.

We'll cover about 40 years worth of technology, skewing toward 2010+

- Current software will probably be obsolete in a few years
- The underlying PRINCIPLES don't change quite so rapidly!

⇒ Get proficient with concepts and current tools, and **learn to adapt!**

## This class in the curricular context

- Introduction to Data Science (DS-GA 1001): Introducing the big concepts, but implemented with relatively small datasets.
- Big Data (DS-GA 1004): Revisiting these concepts, but scaled up with large datasets, and with a focus on implementation to address the specific problems that come with scale.
- So 1001 → 1004 is basically a sequence.

# What should you get out of this class?

- Familiarity with distributed storage and computation
- Appreciation for the technical challenges of big data
- **Understanding of when to use which methods and tools**

Git  
SQL  
Hadoop  
MapReduce  
HDFS  
Spark  
Dremel  
Parquet  
Dask  
CUDA...

Again:

This subject matter involves a lot of obtuse terminology and buzzwords.

**Don't worry.**

If terms are ever unclear, stop and ask for clarification.

**Relatedly:** some of you undoubtedly have more experience than others.

Be mindful of others and the environment we create in the classroom!

We'll demystify a lot of this jargon, but beware:  
A lot of shenanigans can be hidden with jargon,  
souring some on the integrity of the whole field:



**HOW does  
this class  
work?**

# The teaching staff

Instructor

Pascal Wallisch, PhD

Section leaders

Tanvi Bansal

Iris Lu

Tom Zhou

Dennis Hu

Tutor

Jason Moon

Course coordinator

Adeet Patel

Graders

numerous & anonymous

We will use “Brightspace” as the learning management system (LMS) for this class

It will feature

- Announcements
- Lecture slides
- Datasets
- Code
- Assignments
- Assorted class materials (readings, videos, etc.)

You can access it at

<https://brightspace.nyu.edu/>

The *sittyba*

Some highlights worth emphasizing...

# Technology and resources

- All resources are available through **brightspace.nyu.edu**
  - Course schedule, assigned reading, etc.
  - Slides will be posted (usually) immediately before class
- HW assignments will be available via GitHub Classroom
  - If that's a problem, we can make other arrangements
  - Lab section will cover how exactly this works (starting **\*this\*** week)

# Readings

- Each week will have assigned reading, listed in the sittyba
  - Expect a book chapter, or 1-2 papers each week
- Posted under “content” on **brightspace.nyu.edu**
- You’re expected to do the reading **before class meets**
  - Learning is most effective when you first encounter new ideas on your own.
  - We can use the class time to clarify difficult or confusing concepts.

# Grading

- 25% Homework assignments
- 25% Capstone project
- 25% Final
- 9% Quizzes
- 16% Low stakes assignments

# Homework assignments (25%)

- 5 ~bi-weekly programming assignments to be completed **in small groups**
  - Lab sections = get help with this assignment from the section leader
- You'll get access to NYU's high-performance computing (HPC) cluster
- You have **1 grace day** to use for each of these homework assignments
  - After that, **1% penalty per hour** for late submissions.
  - No assignments will be accepted more than 5 days late.
  - Grading these assignments is not easy, please be mindful of the graders' time!
  - Do not plan on using the grace day – this is intended for emergencies (e.g. cluster crash)

## Capstone project (25%)

- This will be an extended homework / programming assignment over 3-4 weeks, integrating several of the tools and methods that we'll cover.
- Due at the end of the semester
- Same rules re late assignments (we are being overly lenient)
- Details will be posted in April

# Final Interview & Skills Test (25%)

- Cumulative, in-class, 64 T/F questions only
- Modest A&I penalty (so EV is still positive, and corresponds to 4 option MC exam)
- Beyond this class, you'll be expected to be conversant in these topics
  - (Think: job interviews!)
- All straight from the lecture. Quizzes will give you a good idea of what to expect
- Date / room and details will be posted on Albert by mid-semester (once the registrar tells us), but it will be during finals week. **Make travel plans accordingly.**

## Quizzes (9%)

- In-person quizzes, during the labs. We count the highest 9 for credit.
- Quizzes are closed book and must be **completed independently**.
- Quiz will be paced during the lab. Don't be latest
- Your lowest quiz scores are automatically dropped.
  - There will not be “make-up” quizzes, but this avoids penalties if you miss one (or 2 or 3).

# Glossary (from my machine learning class, lecture 1)

- Vector
- Scalar
- Column vector
- Row vector
- Transpose
- Vector addition
- Scalar multiplication
- Vector space
- Neutral element
- Zero vector
- Vector subspace
- Linear combination
- Span
- Linear independence
- Basis vectors
- Basis
- Standard basis
- Matrix
- Matrix Dimensionality
- Square matrix
- Symmetric matrix
- Diagonal matrix
- Identity matrix
- Linear transformations
- Linear mappings
- Injective mapping
- One-to-one mapping
- Surjective mapping
- Onto mapping
- Bijective mapping
- Invertible mapping
- Isomorphic mapping
- Automorphic
- mapping
- Identity mapping
- Kernel
- Null space
- Image
- Range
- Column space
- Domain
- Codomain
- Dot product
- Scalar product
- Inner product
- Hadamard product
- Magnitude
- Vector length
- Euclidian norm
- Unit vector
- Norms
- $L_1$  norm
- $L_2$  norm
- Collinear vectors
- Orthogonal vectors
- Projection
- Hyperplane
- Matrix multiplication
- Orthogonal matrices
- Determinant
- Singular matrices
- Tensor
- Cosine similarity
- Affine subspace
- Matrix inverse
- Linear coordinates
- Pseudoinverse

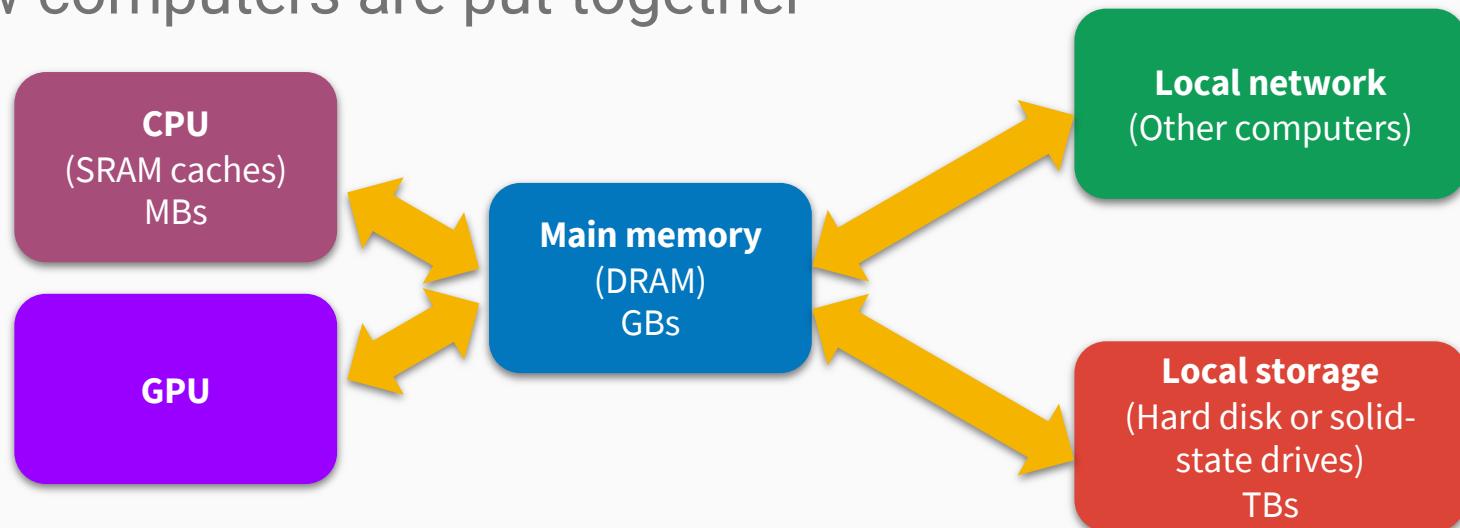
# Academic integrity

- See sittyba for the detailed statement.
- You are responsible for all submitted materials.
- Yes, bots exist, but you still need to understand the material, so will not be allowed to use them during the final.
- **We take academic integrity seriously in this class.**
- Number of people reported for academic integrity violations in this class:
- 2021: 28
- 2022: 14
- 2023: 32 (2 of them twice, some expelled)
- 2024: 0

# Big Data in the broader CS context

# Some basic computer organization

- To know what counts as “too big”, we’ll need to understand how computers are put together



# So: What are our resources?

- **Storage**

- Where and how is data kept?
- How much data can we keep?

- **Communication**

- How quickly can we move data between locations?  
(Eg: over network, disk→RAM, RAM→CPU, RAM→GPU)

- **Computation**

- How quickly can we process data? (Convert *inputs* into *outputs* via *instructions*)

# 1) Storage: The cost of storage over time...

**\$3398 = \$11,292.77  
in 01/2024**

**XCOMP** introduces a complete micro-size disk subsystem with more...

- MORE STORAGE
- MORE SPEED
- MORE LIFE
- MORE SUPPORT

5100 users. The XCOMP subsystem is now available with 10 MB of storage, 5 megabytes also available at \$2,898.00. Compare the features and features of any other 5½-inch — or even 8-inch system, and you'll agree that XCOMP's value is unbeatable.

**OUTPERFORMS OTHER HARD DISKS**  
Floppy disk and larger, more expensive hard disks are no match for the XCOMP subsystem. More data is available on every seek: 64K on 10MB and 32K on 5MB. Faster seek time too — an average of 70MS. It provides a fast, reliable, and ruggedized system with 20 watts of power. Data is protected in the sealed enclosure, and the landing zone for heads provides added protection. The XCOMP electrical power cord plugs directly into the S100 bus and provides power for the drive.

**FAST CONTROLLER**  
The XCOMP controller is the key to this system's high efficiency operation. Speed-up features include intelligent cache, direct block addressing, built-in controller buffer, and read lookahead. OEMs worldwide have already proven the outstanding performance of the XCOMP controller.

**MORE SOFTWARE**  
Included with the system is software for testing, formating, I/O drivers for CP/M™, plus an automatic Disk Diagnostic program. Software drivers for MP/M® and DOS® are also available. The sophisticated formating program assigns alternate sectors to bad sectors detected during formating, assuring the lowest possible error rate — at least ten times better than floppies.

**WARRANTY**  
The XCOMP has a full one-year warranty on parts and workmanship.

**ALSO AVAILABLE FROM XCOMP**

- General Purpose controllers (8 bit interface), with easy interface to microprocessor-based systems.
- GP controller adapter that plugs directly into most S100 boards.
- ST/R GP controller for the 5MB and 10MB drive, above, with ST506 type interface.
- ST/R GP controller for the A1000 interface.
- SW/R GP controller for storage module drives.
- ST/R, 5MB, and 10MB, same as above, for the S100 bus.

Quantity discounts available. Distributor, Dealer, and OEM inquiries invited.  
See your local Dealer, or call:  
**XCOMP, Inc.**  
1560 Torrey Street  
San Diego, CA 92121  
Tel. (714) 271-6730  
Telex: 162766

Circle 425 on Inquiry card.

Byte Magazine, Vol. 6, No. 8 (1981/08)

# 1) Storage: The cost of storage over time...



\$3398 = \$11,292.77  
in 01/2024

x 2 million



x27  
= 540TB

WD Ultrastar HC560 WUH  
SE SATA HDD 0F38785

★★★★★ 12

\$409<sup>15</sup>

FREE delivery Jan 25 - 29

Add to Cart

More Buying Choices

\$359.99 (25 used & new offers)

Byte Magazine, Vol. 6, No. 8 (1981/08)

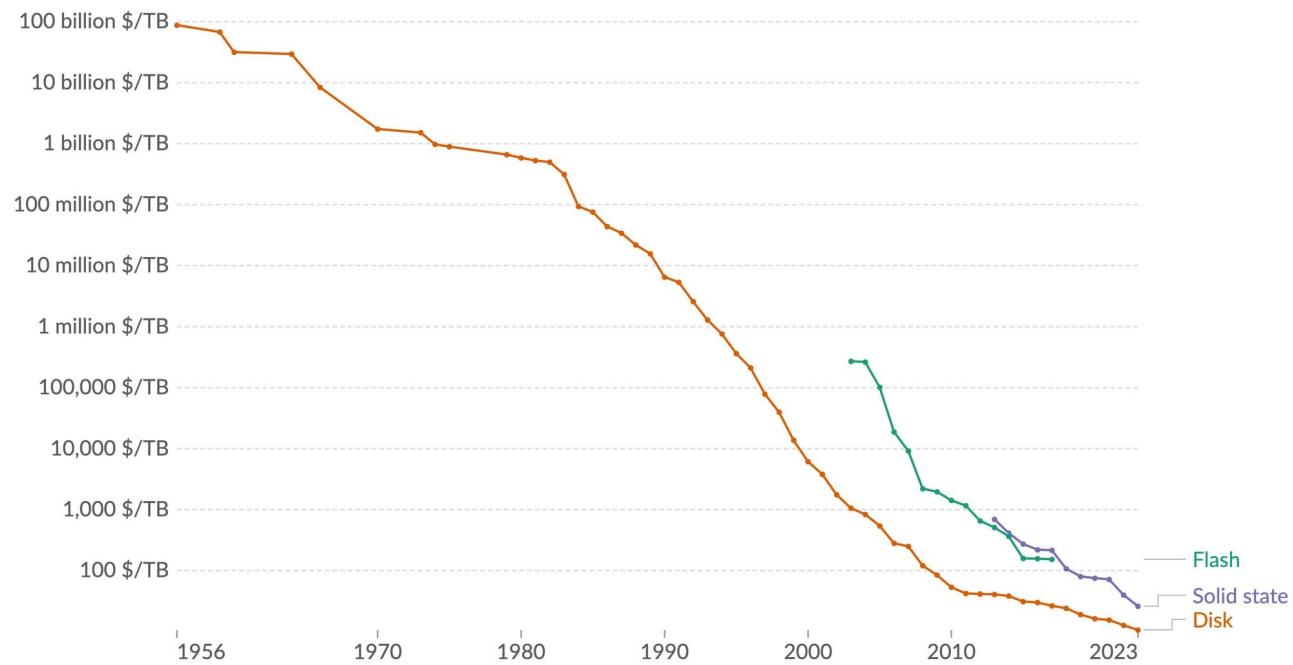
westerndigital.com (2024/01)

# This just keeps going! (The price of storage is usually NOT the bottleneck)

## Historical price of computer storage

Our World  
in Data

Expressed in US dollars per terabyte (TB), adjusted for inflation. "Disk" refers to magnetic storage, "flash" to memory used for rapid data access and rewriting, and "solid state" to solid-state drives (SSDs).



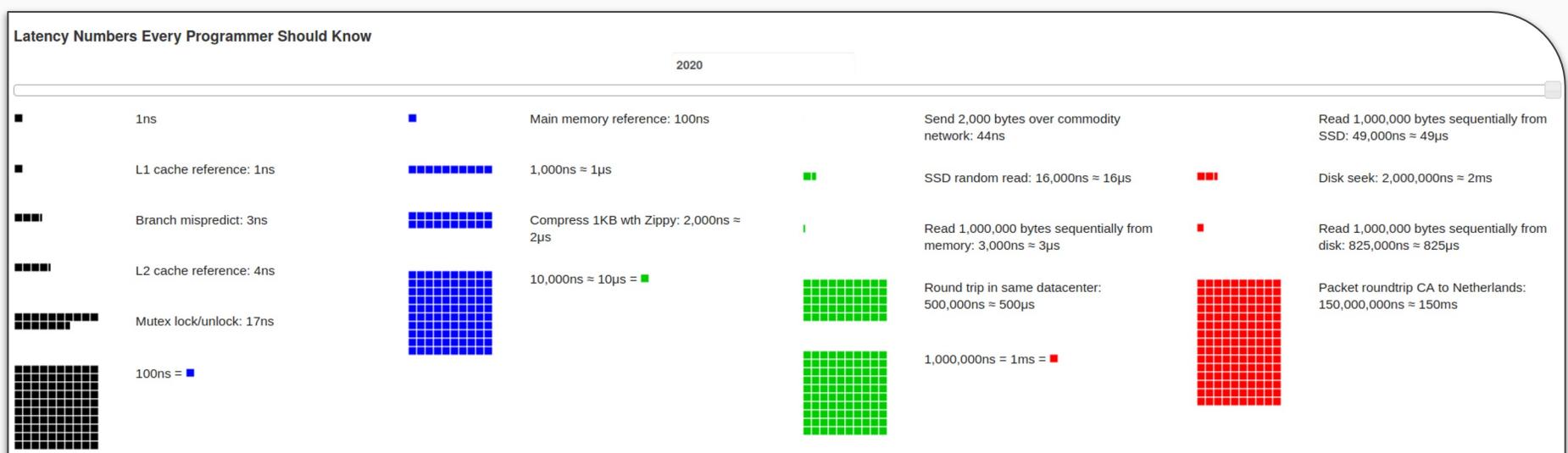
Data source: John C. McCallum (2023); U.S. Bureau of Labor Statistics (2024)

OurWorldInData.org/technological-change | CC BY

Note: For each year, the time series shows the cheapest historical price recorded until that year. This data is expressed in constant 2020 US\$.

## 2) Data volume and velocity

- (2023) YouTube adds 500 hours of HD video every minute  
<http://www.businessofapps.com/data/youtube-statistics/>
- (2019) Facebook generates 4 Petabytes (4e15 bytes) of new data per day  
<https://research.fb.com/facebook-s-top-open-data-problems/>
- (2017) Twitter stores >500PB of data  
[https://blog.twitter.com/engineering/en\\_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html](https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html)
- (2017) CERN data center: > 200PB  
<https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone>



L1 cache read	1 ns
L2 cache read	4 ns
Main memory read	100 ns
SSD random read	16,000 ns
HDD random read	2,000,000 ns

- Pre-fetching and pipelining can **accelerate sequential reads**
- **Main memory** is typically the first bottleneck
- The less communication (read/write) we have to do, the better

### 3) Communication

[https://people.eecs.berkeley.edu/~rcs/research/interactive\\_latency.html](https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)

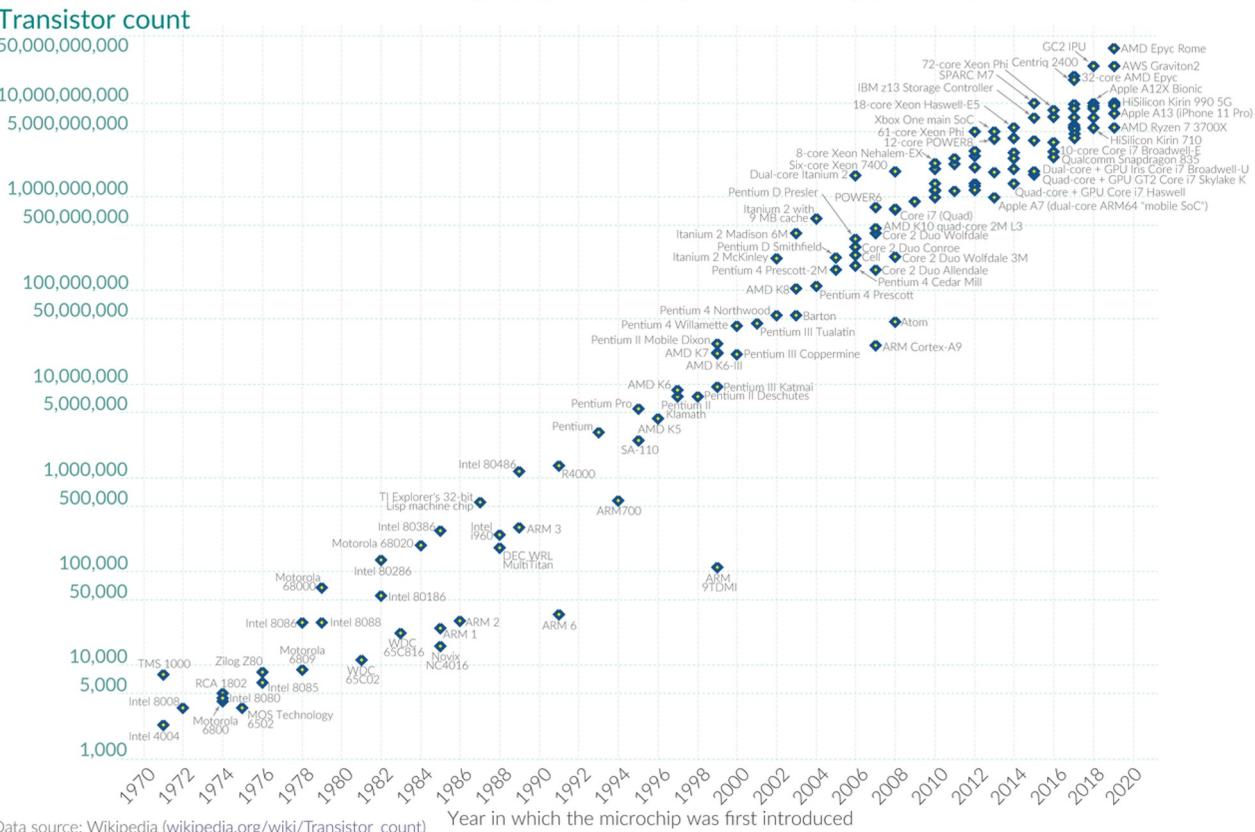
## 4) Computation

**Note: transistor count ≠ speed!**

### Moore's Law:

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World  
in Data

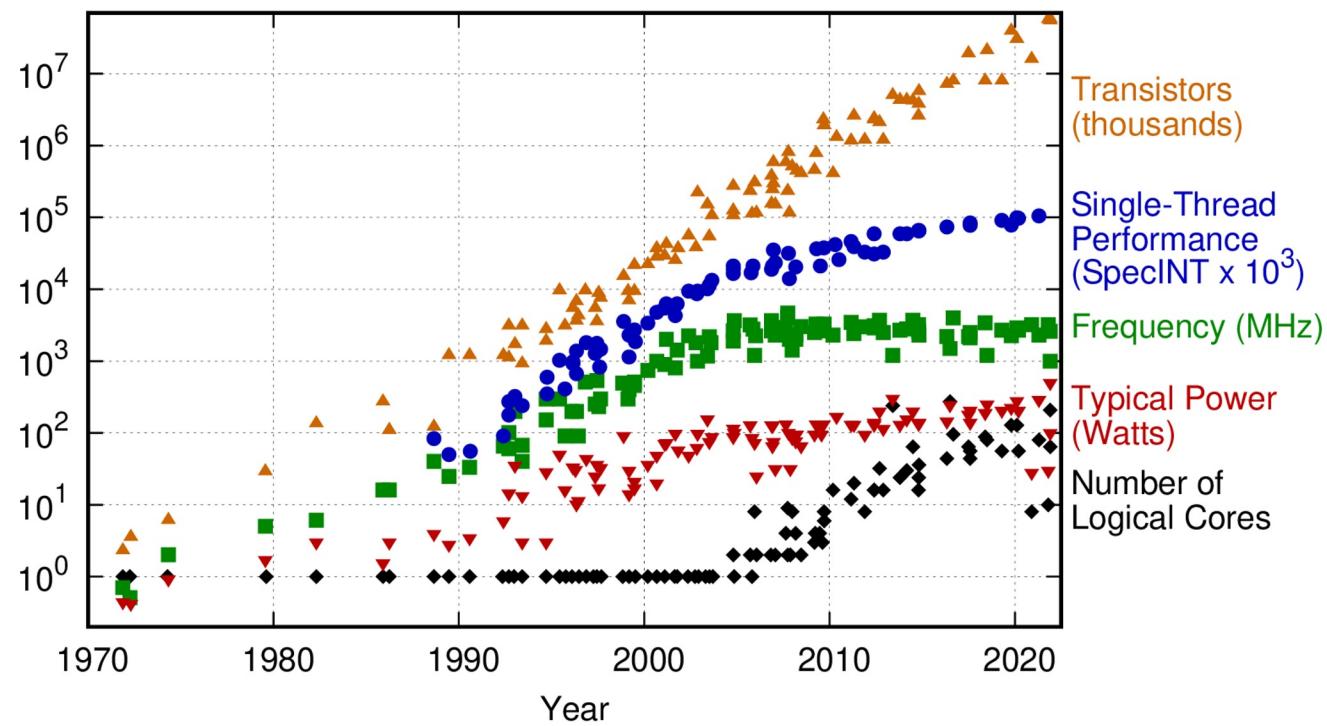


Gordon Moore (1965, 1975)

Moore in 2015:

*"I see Moore's law dying here in the next decade or so."*

50 Years of Microprocessor Trend Data



Gains now come from **parallelism** (multi-core), not **clock speed!**

To summarize -  
what is the  
upshot?

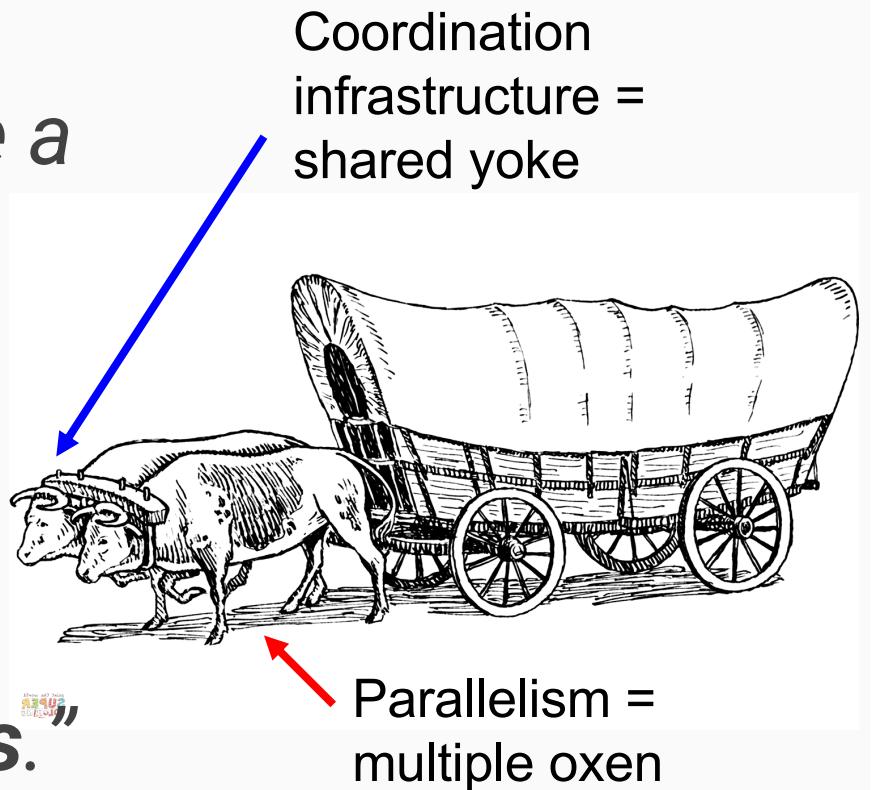
- Storage capacity (per \$) continues to increase
- However, data velocity is also ever increasing
- But CPU speed is \*not\* keeping up
- Communication is even worse (slower), avoid it

## What to do?

## What is the solution?

*“In pioneer days, they used oxen for heavy pulling, and when one ox couldn’t budge a log, they didn’t try to grow a bigger ox.*

*We shouldn’t be trying for bigger computers, but for more **systems** of computers.”*



Rear Admiral Grace Hopper (1906-1992)

Parallelism / load distribution can also help with speed



WHY “communication” is the  
common enemy of all mankind  
(in the context of Big Data)

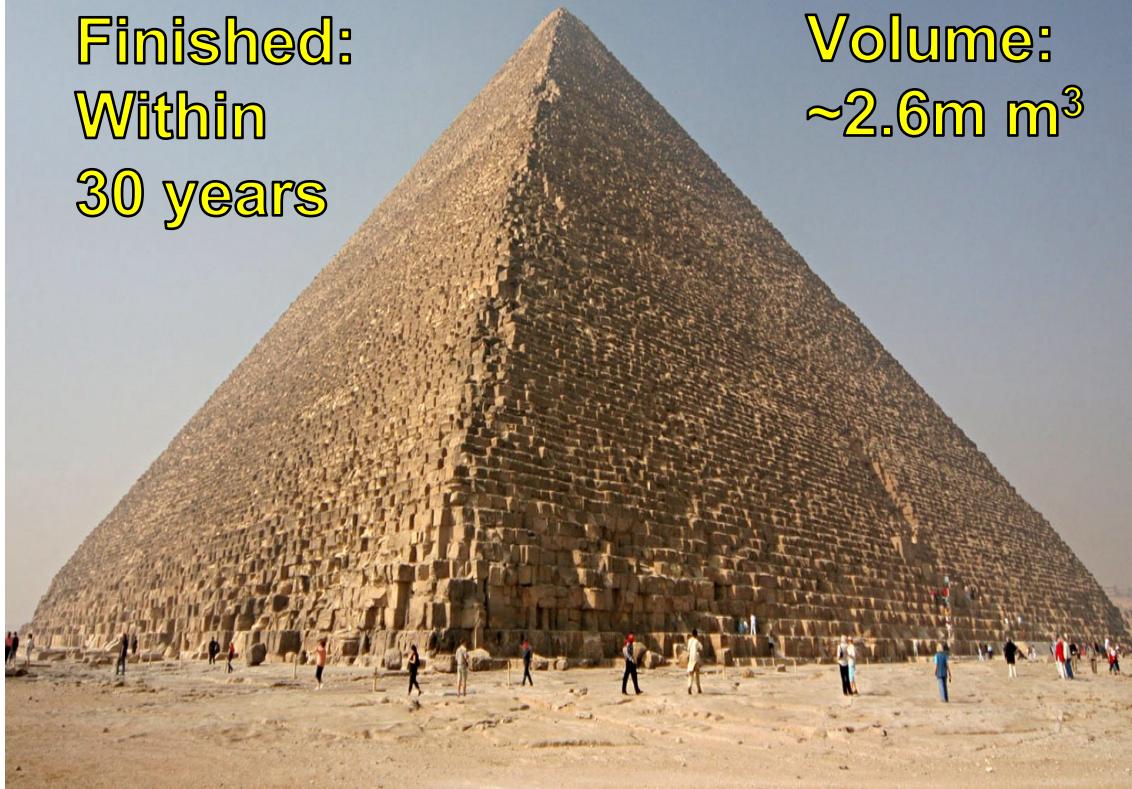
## The Great Pyramid of Giza

Height: 481 ft

Begun: ~2600 BC

Finished:  
Within  
30 years

Volume:  
 $\sim 2.6 \text{m}^3$



## Cologne Cathedral

Height: 515 ft

Begun: 1248

V:  $\sim 0.5 \text{m}^3$

Finished:  
1880



# Principles of scaling: At the seashore



On average, Suzy collects  
100 seashells per hour

Valid  
?

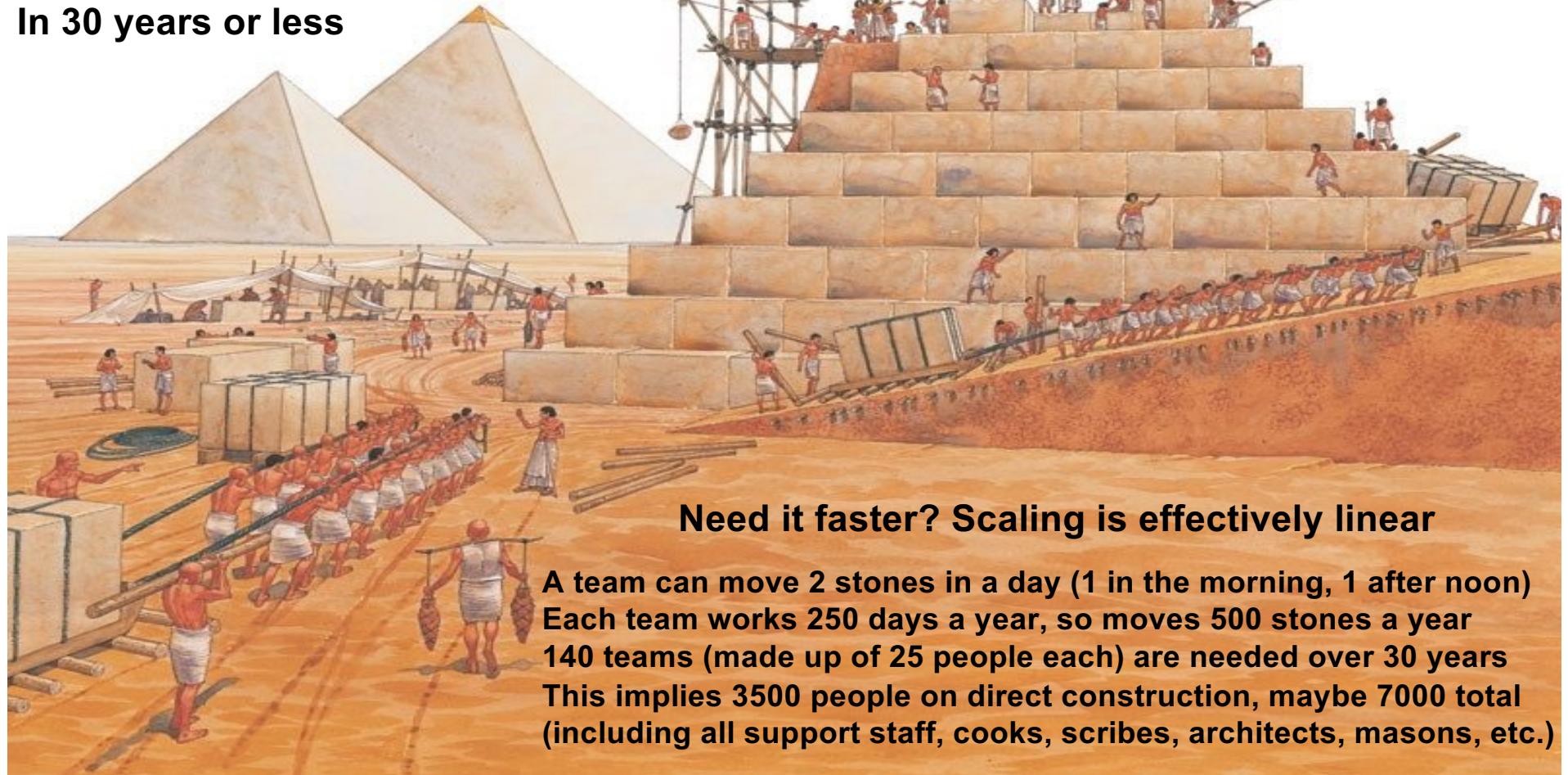


Together, Suzy and Lucy  
collect 200 seashells per hour

**Some problems are easy to parallelize:**

**~2 million stones need to be moved**

**In 30 years or less**



**Need it faster? Scaling is effectively linear**

A team can move 2 stones in a day (1 in the morning, 1 after noon)

Each team works 250 days a year, so moves 500 stones a year

140 teams (made up of 25 people each) are needed over 30 years

This implies 3500 people on direct construction, maybe 7000 total  
(including all support staff, cooks, scribes, architects, masons, etc.)

In other words, the Pyramid of Giza took about 2.5 million “man-months” of human effort to build (all people on all teams over all years of construction).

A modern operating system typically consists of about 100 million lines of code

A professional software engineer can write about 10,000 production level (unit tested, etc.) lines of code a year (40/day)

Can a major corporation simply hire 10,000 software engineers and finish the project within a year?

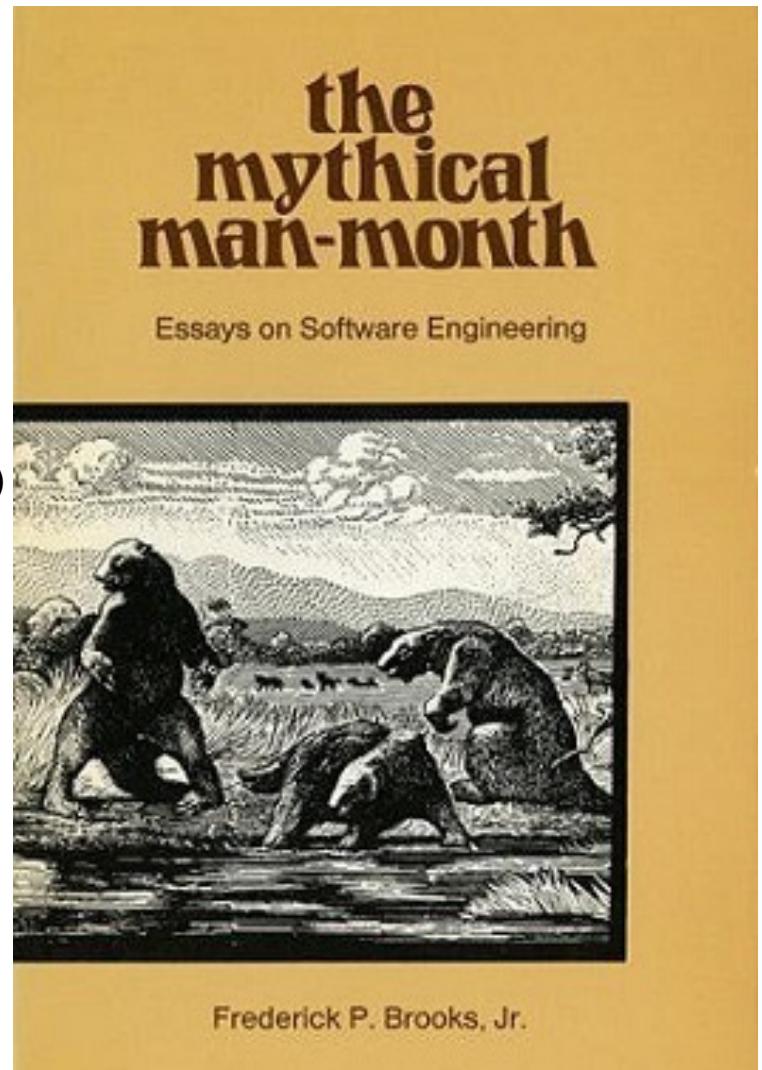
No!

But why not?

Assembling the blocks of code is not independent!

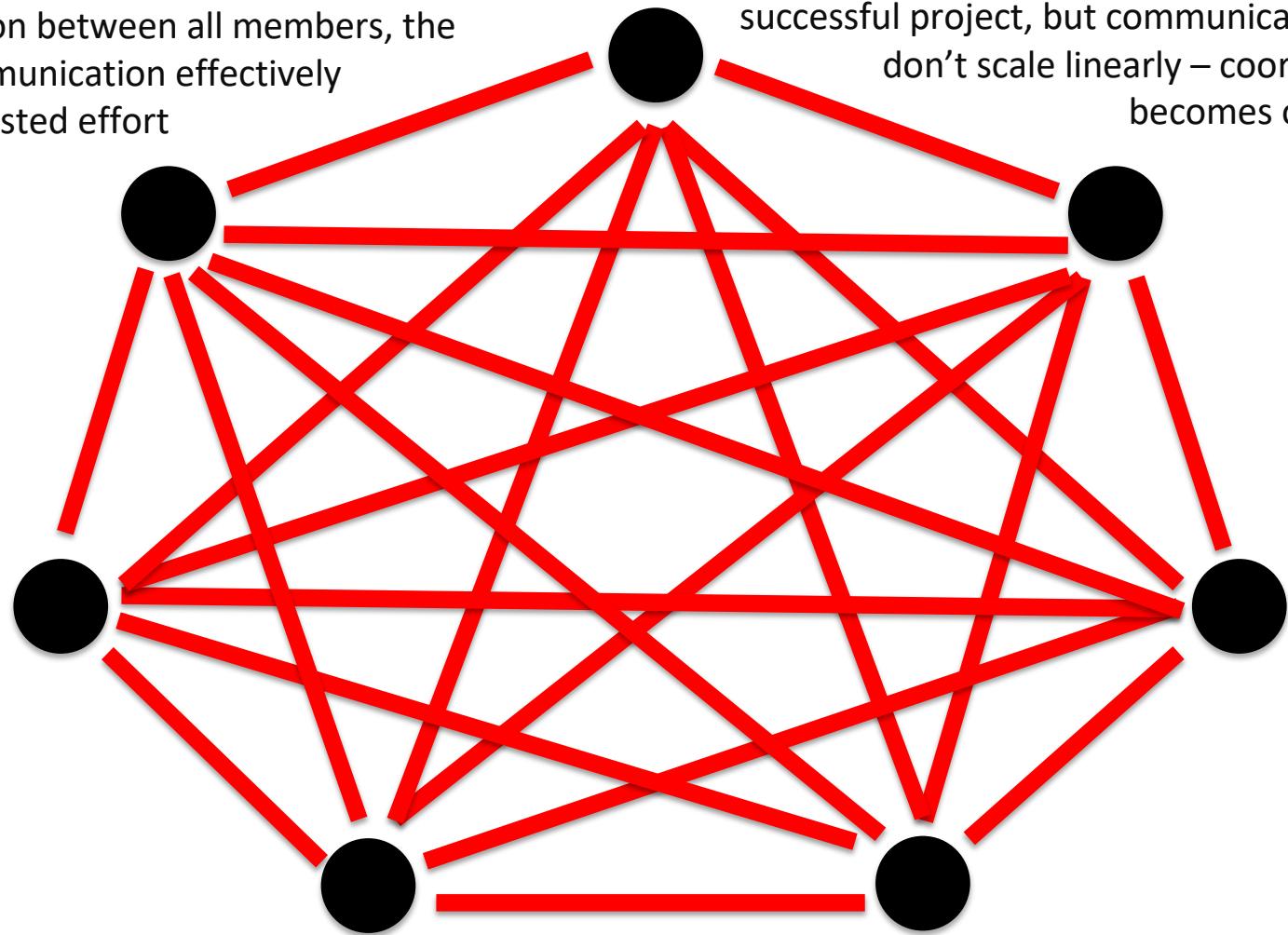
*“Adding manpower to a late software project makes it later.”*

But why?



If the success of a (complex) project depends on coordination between all members, the need for communication effectively represents wasted effort

Communication might be necessary for a successful project, but communication demands don't scale linearly – coordination soon becomes overwhelming



The upshot:  
Parallelizing complex tasks like data  
analysis jobs is not trivial

- We will distribute processing
- We will distribute storage
- While minimizing communication and coordination,  
wherever possible
- We might have to impose processing hierarchies...

# Next time...

Centralized systems:  
File systems  
The relational model  
and databases

- Reading for next week  
[Garcia-Molina, Ullman, & Widom, 2009, ch2]



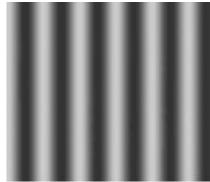
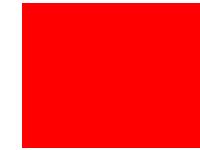
Smallest font



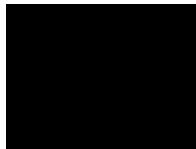
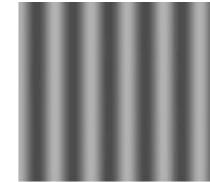
Please turn off and put  
away your cell phone



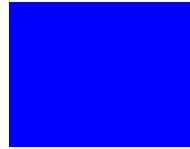
# Calibration slide



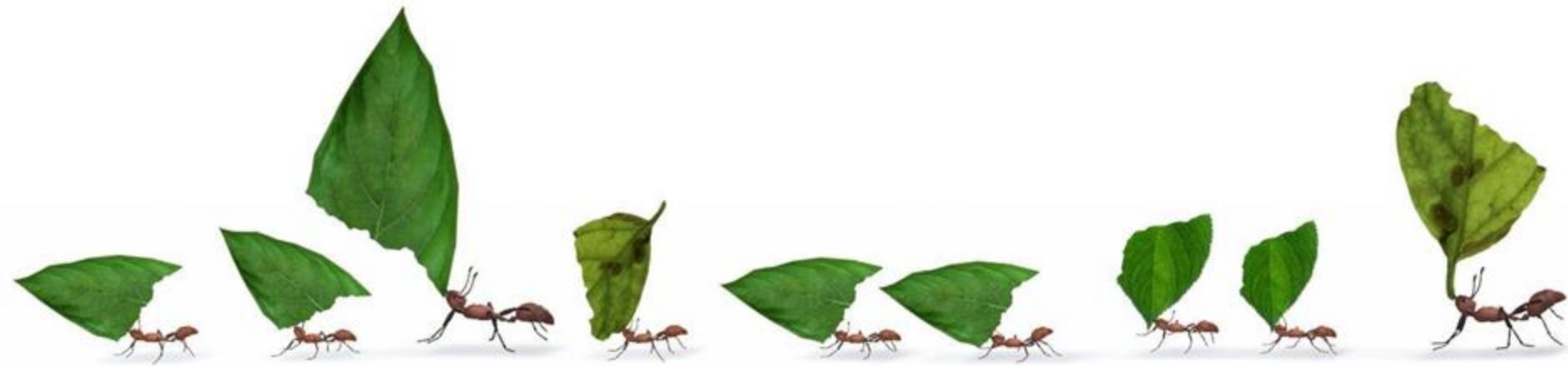
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



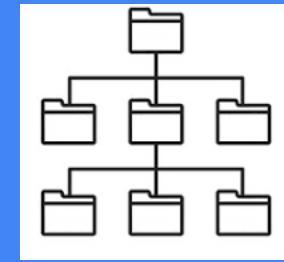
# Big Data



# Week 02: Centralized systems

File systems

Relational databases



DS-GA 1004: Big Data

# Announcements

- This week: Lab 2 (SQL)
- The office hour schedule of all teaching staff is now in effect
- Download the sittyba today for an updated version (all office hours, accurate due dates, working links, etc.)
- **HW 1 is due next Monday (02/10)**

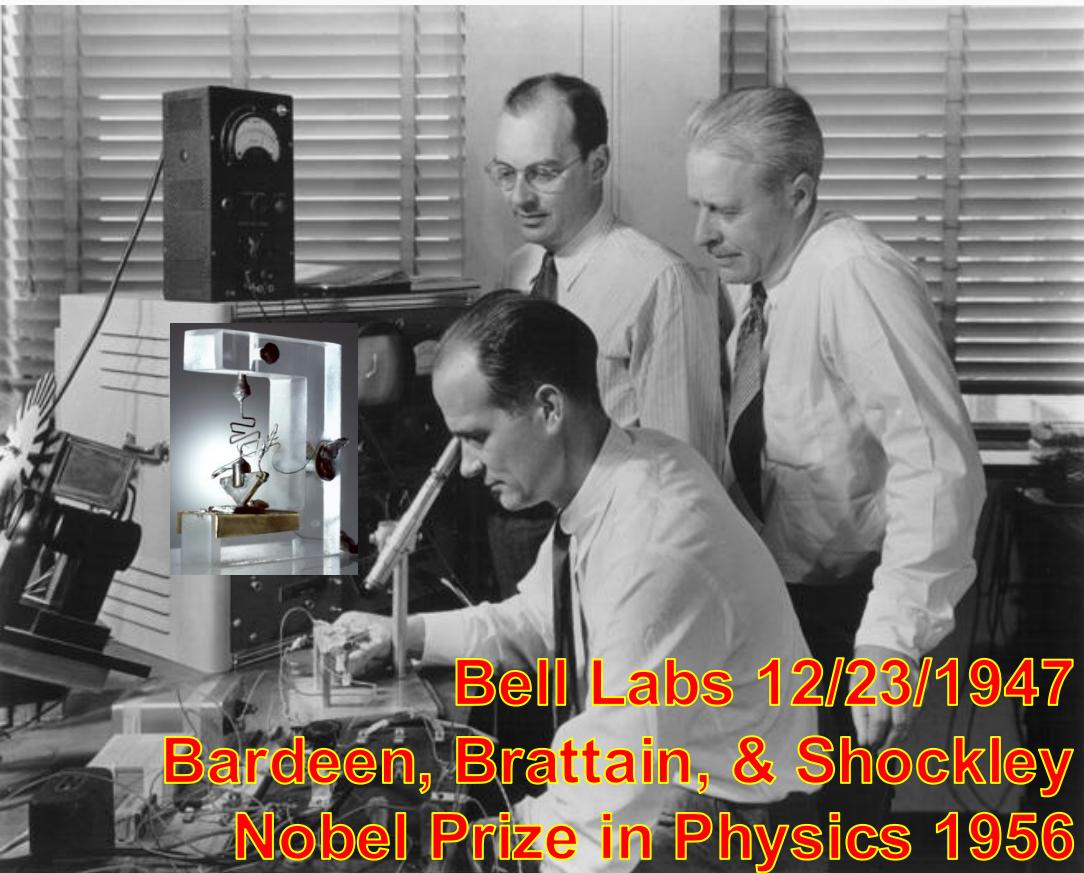
# Confusion, Doubt, & Struggle (CDS)

1) Is a large array that came from a random number generator “data” (for the purpose of this class)?

221	46	168	42	36	165	40	207	83	116	119	103	63	197	127	96	18	227	86	208	5	200	34
37	65	175	35	48	228	157	44	76	133	76	254	39	3	208	51	226	165	25	55	166	140	129
241	155	4	188	254	12	232	206	168	176	211	148	42	124	97	201	251	187	104	55	240	150	140
18	142	58	115	236	82	195	70	177	186	174	222	172	230	131	127	162	39	79	3	131	145	108
41	29	10	77	43	115	45	192	173	208	112	252	52	1	62	13	60	55	198	206	255	110	14
89	79	193	76	181	99	161	247	248	67	16	120	3	121	203	70	111	200	44	48	203	143	204
39	113	0	122	30	136	75	37	33	232	181	4	52	201	212	246	22	90	28	133	51	145	142
181	118	76	217	39	220	176	148	152	184	226	123	110	199	193	247	16	12	228	177	85	99	169
30	178	154	88	89	7	25	221	21	184	96	156	216	237	38	125	20	178	96	91	18	47	242
7	216	71	28	223	201	223	119	208	212	167	81	113	155	137	71	243	36	241	55	216	18	219
185	86	233	199	77	37	149	88	248	139	104	157	125	251	19	250	58	118	21	2	212	171	71
152	186	153	50	238	92	33	150	182	209	202	132	227	4	117	59	203	131	185	195	110	12	181
57	20	111	211	189	20	224	12	212	155	48	65	138	202	207	36	32	54	155	138	110	231	126
129	2	225	103	71	255	155	181	72	165	249	31	123	79	32	242	62	236	254	207	229	66	72
37	234	192	171	161	93	109	122	166	250	149	42	128	105	213	193	90	58	127	114	206	17	169
107	33	102	206	197	69	217	150	169	33	181	252	219	67	193	201	80	83	81	146	54	225	241
101	184	6	195	192	47	210	65	218	138	171	16	63	71	89	210	139	70	158	99	63	174	129
238	52	4	244	180	146	193	170	46	117	130	118	9	183	233	242	19	162	245	178	238	120	1
210	131	245	166	240	162	15	197	228	54	170	195	126	213	195	26	31	141	102	122	69	36	71
146	207	204	121	64	181	122	228	239	163	205	161	12	216	15	226	102	156	85	243	175	225	28
124	103	215	34	97	61	156	3	93	14	162	10	169	37	213	137	87	9	36	161	93	143	150
99	8	238	208	169	203	82	184	231	112	210	42	10	103	102	45	217	154	72	72	14	242	78
104	96	91	45	203	20	162	68	145	190	119	198	172	254	146	68	130	176	69	202	14	34	224
162	246	111	88	130	149	89	34	226	185	121	171	224	204	139	192	121	70	85	187	44	104	37
163	230	133	87	54	208	148	70	116	192	28	60	206	105	233	92	197	211	240	143	12	248	144
7	181	227	145	18	76	130	46	94	102	221	90	113	23	125	17	43	164	227	179	92	93	233
158	68	51	150	45	42	199	232	230	179	212	6	175	80	55	17	243	219	214	215	135	135	231
41	123	86	29	10	41	49	229	204	10	153	133	122	217	248	214	98	216	21	225	141	239	158
122	61	191	200	53	111	107	159	190	140	46	233	43	0	244	67	47	137	177	47	178	188	61
123	202	32	15	239	70	81	189	213	53	88	195	81	171	223	3	64	123	16	110	97	64	186
37	213	24	182	117	94	55	144	226	32	219	163	211	106	141	236	62	11	192	63	171	51	215
92	111	0	77	139	226	216	126	114	128	156	216	40	125	40	178	87	209	158	192	1	117	146
202	141	123	64	7	36	254	43	195	46	225	27	157	243	116	101	94	232	180	199	121	234	132
164	127	94	40	58	106	90	138	160	193	165	241	118	175	159	224	31	209	164	227	147	57	110
32	223	182	83	180	43	106	34	149	7	34	247	7	208	110	226	216	225	150	251	171	198	101
232	113	212	86	142	57	99	213	194	179	85	16	231	4	161	69	17	105	78	245	8	193	215
187	125	154	10	215	176	97	101	164	85	145	143	139	209	19	106	183	164	104	34	136	78	195
7	83	106	153	167	164	145	241	108	127	159	19	5	123	200	77	72	75	24	232	253	88	213
254	230	83	212	6	201	100	196	53	76	130	71	234	165	14	41	73	196	148	119	190	56	193
41	45	234	96	140	229	111	189	1	144	127	85	250	247	190	42	173	174	54	117	58	187	100
131	14	44	224	99	238	48	215	147	217	110	181	114	98	111	67	220	91	207	187	116	81	118
19	13	250	31	103	220	158	214	130	182	179	214	199	214	207	217	221	249	58	98	176	241	206

# Confusion, Doubt, & Struggle (CDS)

2) What is a transistor and why does it matter how many of them there are on a chip?

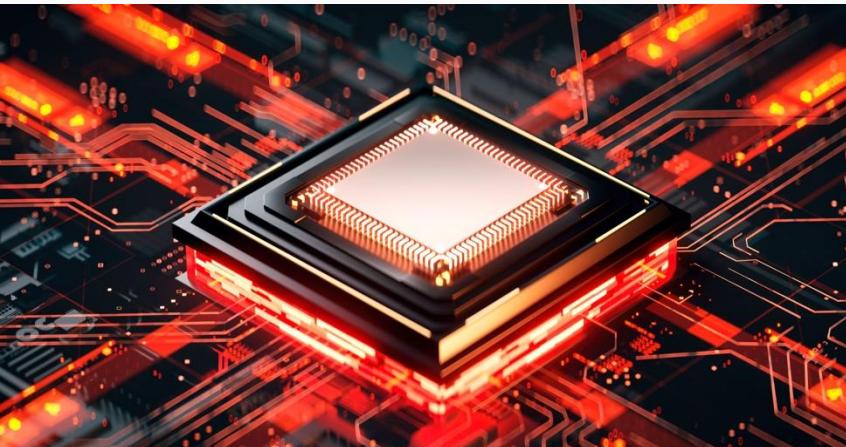


**Bell Labs 12/23/1947**  
**Bardeen, Brattain, & Shockley**  
**Nobel Prize in Physics 1956**

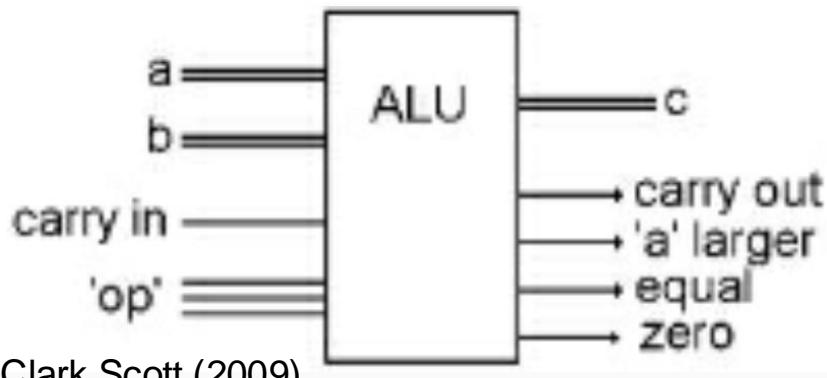
- Transistors are very fast and reliable electronic switches.
- They allow currents to either flow in a wire or not, implementing binary logic.
- Since their invention, it has been possible to miniaturize them by several orders of magnitude (modern transistors = single digit nm), allowing to pack billions of them on a chip.
- The physical implementation is beyond this class, but concisely – when a voltage is applied to a “doped” semiconductor (usually germanium or silicone) in a transistor, it effectively opens a gate that allows current to flow (otherwise it remains closed).

# Confusion, Doubt, & Struggle (CDS)

## 3) What is a CPU (central processing unit) aka “the processor”?

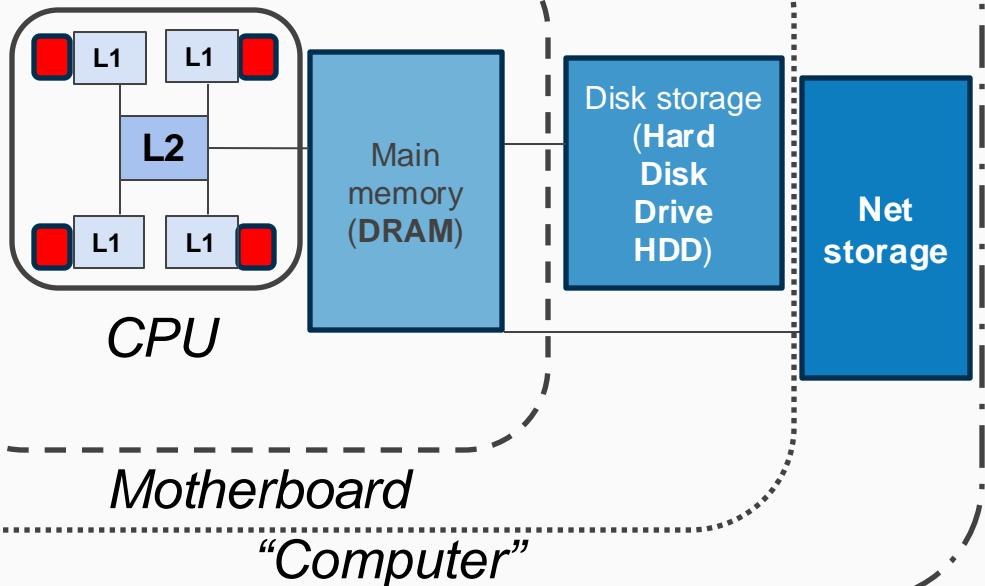


- The CPU is the “brain” of the computer.
- It consists of several components, including many transistors that are arranged in an “integrated circuit”.
- CPUs store a little bit of data locally (in the L1/L2 caches) and – together with the instructions (from programs) process them, which results in outputs.
- The clock of the CPU generates a signal that synchronizes CPU operations (several billion times a second in modern CPUs).
- Each clock cycle, a CPU (core) can perform one operation, e.g. reading data from memory, executing instructions or writing data to memory.



# Confusion, Doubt, & Struggle (CDS)

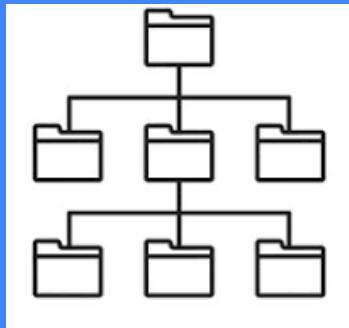
## 4) How does a (modern) computer work?



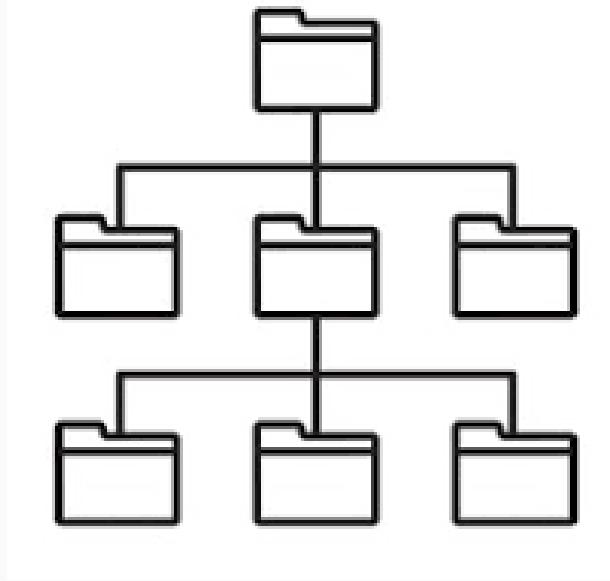
- Why does this matter?
- In a computer, electrical signals travel down wires at a substantial fraction (50-75%, depending on material and temperature) of the speed of light in vacuum.
- The speed of light in vacuum corresponds to a distance just shy of a foot ( $11.8'' = 30 \text{ cm}$ ) per ns.
- As signals go back and forth to implement the computation, this can add up.
- Less distance = faster execution.
- Distance is time

Schematic: Not to scale, there are other architectures.  
Not depicted: Non class-relevant components (e.g. fan, power,...)

# Today: 4 parts



- File systems
- Relational databases
- SQL
- Transaction integrity



# File systems!

- To understand modern distributed tools, it helps to understand the solutions of previous generations.
- What problems did people face in designing:
  - Relational databases?
  - Map-reduce?
  - Spark?
  - Dask?

# File systems

	003_Mozart_RequiemInDMinor_Section4_15.mp4
	003_Mozart_RequiemInDMinor.mp3
	004_Beethoven_FurElise_Section1_05.mp4
	004_Beethoven_FurElise_Section1_10.mp4
	004_Beethoven_FurElise_Section1_15.mp4
	004_Beethoven_FurElise_Section2_05.mp4
	004_Beethoven_FurElise_Section2_10.mp4
	004_Beethoven_FurElise_Section2_15.mp4
	004_Beethoven_FurElise_Section3_05.mp4
	004_Beethoven_FurElise_Section3_10.mp4
	004_Beethoven_FurElise_Section3_15.mp4
	004_Beethoven_FurElise_Section4_05.mp4
	004_Beethoven_FurElise_Section4_10.mp4
	004_Beethoven_FurElise_Section4_15.mp4
	004_Beethoven_FurElise.mp3
	005_Debussy_ClaireDeLune_Section1_05.mp4
	005_Debussy_ClaireDeLune_Section1_10.mp4
	005_Debussy_ClaireDeLune_Section1_15.mp4
	005_Debussy_ClaireDeLune_Section2_05.mp4
	005_Debussy_ClaireDeLune_Section2_10.mp4
	005_Debussy_ClaireDeLune_Section2_15.mp4
	005_Debussy_ClaireDeLune_Section3_05.mp4
	005_Debussy_ClaireDeLune_Section3_10.mp4
	005_Debussy_ClaireDeLune_Section3_15.mp4
	005_Debussy_ClaireDeLune_Section4_05.mp4
	005_Debussy_ClaireDeLune_Section4_10.mp4
	005_Debussy_ClaireDeLune_Section4_15.mp4
	005_Debussy_ClaireDeLune.mp3

- You should all (?) be intuitively familiar with these
- Use directories to organize your data
- Structured data can be stored as files  
⇒ data persists across application runs

# File systems

	003_Mozart_RequiemInDMinor_Section4_15.mp4
	003_Mozart_RequiemInDMinor.mp3
	004_Beethoven_FurElise_Section1_05.mp4
	004_Beethoven_FurElise_Section1_10.mp4
	004_Beethoven_FurElise_Section1_15.mp4
	004_Beethoven_FurElise_Section2_05.mp4
	004_Beethoven_FurElise_Section2_10.mp4
	004_Beethoven_FurElise_Section2_15.mp4
	004_Beethoven_FurElise_Section3_05.mp4
	004_Beethoven_FurElise_Section3_10.mp4
	004_Beethoven_FurElise_Section3_15.mp4
	004_Beethoven_FurElise_Section4_05.mp4
	004_Beethoven_FurElise_Section4_10.mp4
	004_Beethoven_FurElise_Section4_15.mp4
	004_Beethoven_FurElise.mp3
	005_Debussy_ClaireDeLune_Section1_05.mp4
	005_Debussy_ClaireDeLune_Section1_10.mp4
	005_Debussy_ClaireDeLune_Section1_15.mp4
	005_Debussy_ClaireDeLune_Section2_05.mp4
	005_Debussy_ClaireDeLune_Section2_10.mp4
	005_Debussy_ClaireDeLune_Section2_15.mp4
	005_Debussy_ClaireDeLune_Section3_05.mp4
	005_Debussy_ClaireDeLune_Section3_10.mp4
	005_Debussy_ClaireDeLune_Section3_15.mp4
	005_Debussy_ClaireDeLune_Section4_05.mp4
	005_Debussy_ClaireDeLune_Section4_10.mp4
	005_Debussy_ClaireDeLune_Section4_15.mp4
	005_Debussy_ClaireDeLune.mp3

- You should all (?) be intuitively familiar with these
- Use directories to organize your data
- Structured data can be stored as files
  - ⇒ data persists across application runs
- Some great properties:
  - **Easy** to implement
  - **Data** does not vanish
  - **Inherently** organized (tree structure)
  - **Portable** across systems

# **File systems can be awesome!**

**So why are we not always using them  
(exclusively)?**

# Reasons not to rely (only) on file systems

- Does not expose or exploit the structure of data
  - What if I want to search by file contents?  
Better options than brute force?
- Each query / analysis required writing a new program
  - Little re-usability between similar analyses
  - Or the same analysis with slightly different data structures
- Directory hierarchies may be too restrictive

003_Mozart_RequiemInDMinor_Section4_15.mp4
003_Mozart_RequiemInDMinor.mp3
004_Beethoven_FurElise_Section1_05.mp4
004_Beethoven_FurElise_Section1_10.mp4
004_Beethoven_FurElise_Section1_15.mp4
004_Beethoven_FurElise_Section2_05.mp4
004_Beethoven_FurElise_Section2_10.mp4
004_Beethoven_FurElise_Section2_15.mp4
004_Beethoven_FurElise_Section3_05.mp4
004_Beethoven_FurElise_Section3_10.mp4
004_Beethoven_FurElise_Section3_15.mp4
004_Beethoven_FurElise_Section4_05.mp4
004_Beethoven_FurElise_Section4_10.mp4
004_Beethoven_FurElise_Section4_15.mp4
004_Beethoven_FurElise.mp3
005_Debussy_ClaireDeLune_Section1_05.mp4
005_Debussy_ClaireDeLune_Section1_10.mp4
005_Debussy_ClaireDeLune_Section1_15.mp4
005_Debussy_ClaireDeLune_Section2_05.mp4
005_Debussy_ClaireDeLune_Section2_10.mp4
005_Debussy_ClaireDeLune_Section2_15.mp4
005_Debussy_ClaireDeLune_Section3_05.mp4
005_Debussy_ClaireDeLune_Section3_10.mp4
005_Debussy_ClaireDeLune_Section3_15.mp4
005_Debussy_ClaireDeLune_Section4_05.mp4
005_Debussy_ClaireDeLune_Section4_10.mp4
005_Debussy_ClaireDeLune_Section4_15.mp4
005_Debussy_ClaireDeLune.mp3

## When are file systems not awesome?

- When your data is structured along multiple axes
- When data have complex interactions
- When your analyses are complex
- **Relational databases to the rescue!**

# Databases did \*not\* replace file systems

- File archives are still the most common way to share large datasets
  - But we usually include some metadata/indexing structure as well
- As we'll see soon, **Hadoop** relies on a **distributed** file system
  - DB **abstractions** can be built on top
  - But this comes with **restrictions** on file contents and structure

# File-based storage

- Often, data often lives (permanently) **on disk / in the file-system**
- If it's small, we can load it into **main memory**:
  - `df ← read_csv('my_data.csv')`
  - `analyze(df)`
- If it's **too big**, we have several options:
  - **Sampling** / approximate computation
  - **Stream processing** (one or few records at a time)
  - **Data structures** / index structures
  - **Parallel computation**
  - ...
  - Buy more memory

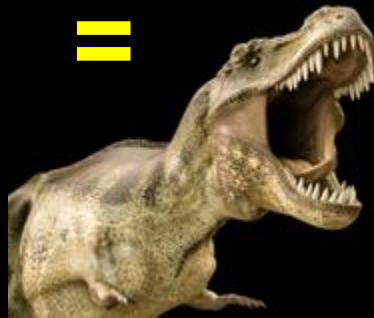
Good solutions often combine two or more of these strategies!

We'll see that often this semester.

# Where we're going next

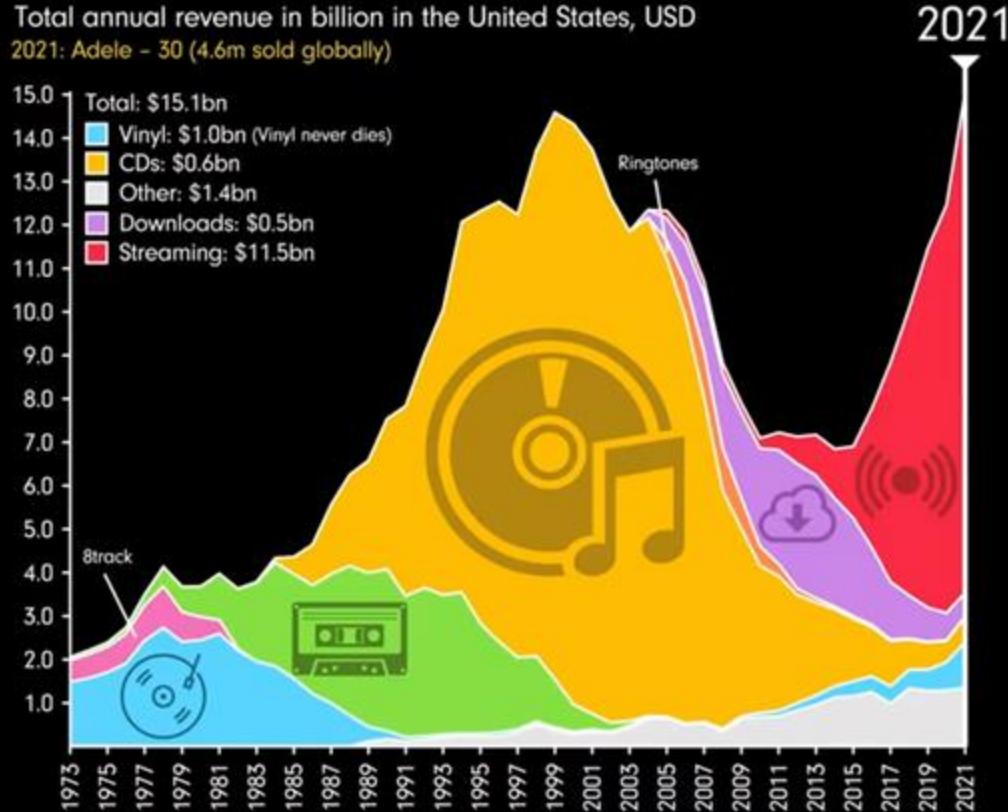
- **Database management systems (DBMS)**
  - Provide a standardized interface to store, load, and process data
- The **relational model** imposes constraints on how data is organized
  - i.e., tables / spreadsheets / dataframes
- Putting these two ideas together = **RDBMS!**

Relational databases are the Vynil  
of data storage and management



Source

## The rise and fall of music formats

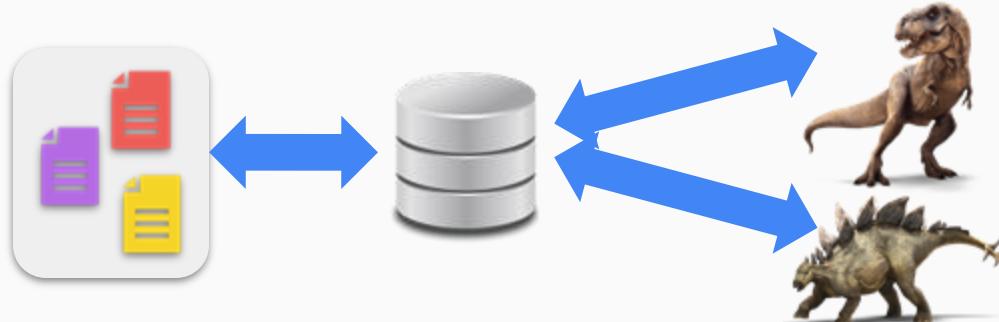


Source: RIAA, Billboard 200, IFPI, ARIA

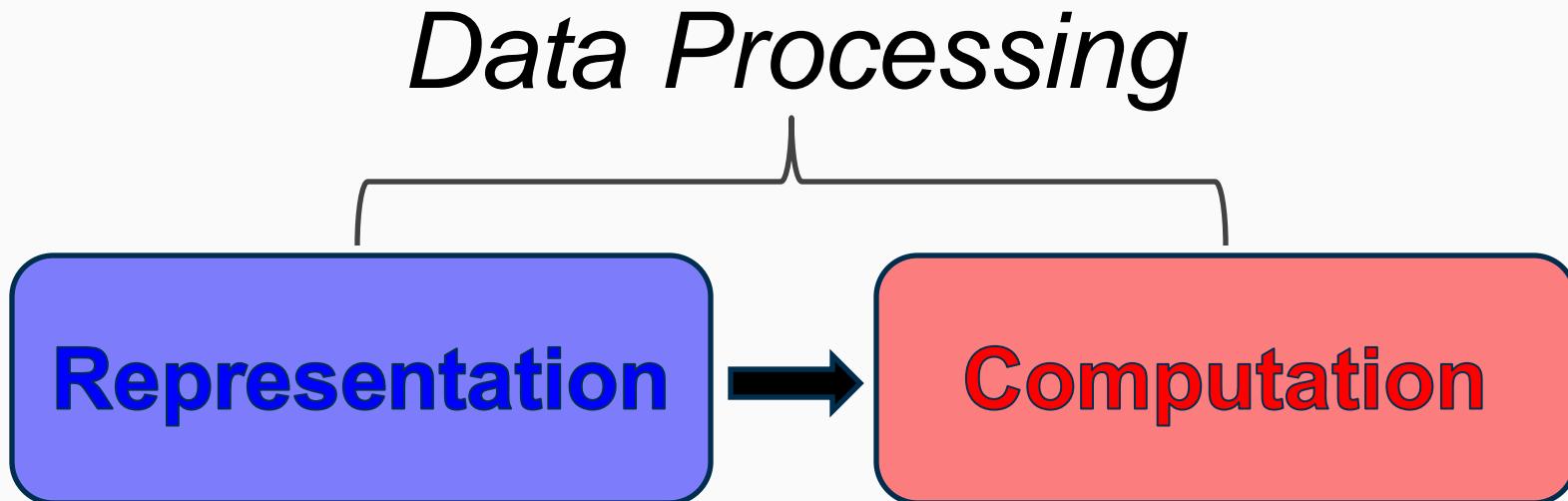
But why?

# Database management systems (DBMS)

- DBMS's job is to provide
  - Data integrity / consistency
  - Concurrent access
  - Efficient storage and access
  - Standardized format / administration
  - Standardized query interface (language)
- DBMS come in many flavors
  - **Relational (RDBMS)**
  - Semi-structured (e.g., XML)
  - Object-oriented
  - Object-relational
  - ...



# The philosophy of the relational model



The relational model constrains  
how the data is represented  
(through the use of schemas)

# The relational model

- **High-level:** tables of data that you're probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.
- Each column of a table represents a **set** of possible values (numbers, strings, etc.)

# The relational model

- **High-level:** tables of data that you're probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.
- Each column of a table represents a **set** of possible values (numbers, strings, etc.)
- A ***relation*** over sets  $A_1, A_2, A_3, \dots, A_n$  is a **subset** of their cartesian product
  - $R \subseteq A_1 \times A_2 \times A_3 \times \dots \times A_n$
  - The **rows** of the table are elements of  $R$ , also known as **tuples**
  - $(a_1, a_2, \dots, a_n) \in R \Rightarrow a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n$

# Example: pachyderms

- $A_1 = \{s \mid s \text{ is a string}\}$   
 $A_2 = \{\text{"Miocene"}, \text{"Pliocene"}, \text{"Pleistocene"}, \text{"Holocene"}, \dots\}$   
 $A_3 = \{\text{"Carnivore"}, \text{"Herbivore"}, \text{"Omnivore"}, \dots\}$   
 $A_4 = \{\text{False}, \text{True}\}$
- Any  $A_i$  could be finite or infinite
- $R \subseteq A_1 \times A_2 \times A_3 \times A_4$  need not contain all combinations!

Species	Era	Diet	Extinct
Elephant	Holocene	Herbivore	False
Woolly Mammoth	Pleistocene	Carnivore	True
Mastodon	Pliocene	Herbivore	True

# Aside: why “relations” and not “tables”?

- Relations are the abstract model of data
- **Table** refers to an **explicitly** constructed relation
  - I.E., records you've observed / collected
- Other relations in a DB:
  - **view**: a relation defined **implicitly**, and constructed **dynamically** at run-time
  - **temporary table**: the output of a **query**

# Exercise: counting relations

Of a set A with 5 elements and a set B with 3 elements (this is not trivial)

- Count the combinations:

$$|A| = 5, |B| = 3 \Rightarrow |A \times B| = 15$$

- # Relations = # Possible subsets of the combinations

$$|2^{A \times B}| = 2^{15}$$

- Order matters!  $A \times B \neq B \times A$ . We need to count both!

$$2^{15} + 2^{15} = 2^{16}$$

- But now we've double-counted the empty relation  $\emptyset \in 2^{A \times B}$  and  $\emptyset \in 2^{B \times A}$   
 $\Rightarrow 2^{15} + 2^{15} - 1 = 2^{16} - 1 = 65535$

# Properties of relations

- $R \subseteq A_1 \times A_2 \times A_3 \times \dots \times A_n$  is a set
  - The tuples (rows) of  $R$  are **unordered**
  - Tuples are **unique**  $\Rightarrow$  no duplicates!
  - Relations over common domains (columns) can be combined by set operations

Species	Era	Diet	Extinct
Elephant	Holocene	Herbivore	False
Woolly Mammoth	Pleistocene	Carnivore	True
Mastodon	Pliocene	Herbivore	True

# Properties of relations

- $R \subseteq A_1 \times A_2 \times A_3 \times \dots \times A_n$  is a set
  - The tuples (rows) of  $R$  are **unordered**
  - Tuples are **unique**  $\Rightarrow$  no duplicates!
  - Relations over common domains (columns) can be combined by set operations
- In practice, add a column (e.g.,  $A_0$ ) with **identifiers** to force uniqueness
  - This is not (usually) part of the data, but is generated automatically by the DBMS
  - ID fields are often used as **primary keys**, and give a default order to rows

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

# Schemas

- A relation is defined by a **schema**:

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Datacene	Data	False

Pachyderms(id: int, Species: string, Era: string, Diet: string, Extinct: boolean)

- Any tuple (int, string, string, string, boolean) is valid under this schema
  - Schemas enforce type (syntax), not semantics!



# Schemas can be hard to design!

- Imagine making a schema to track customers

`Customer(id: int, lastname: string, firstname: string)`

- Are all strings valid as names?
- What constraints could/should you add to the name field of our customer database to ensure data integrity?

# This can go wrong very easily...

- Length constraints are problematic in both directions
- So are character set constraints (accents, spaces, etc.)
- Search and linkage are difficult if the data must be modified to fit schema
- **Be careful!**

# Relational databases

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True

- A relational database consists of one or more relational schemas
- Structured data can be encoded by joining on **shared attributes**
- The collection of schemas defines your **data model**



<b>id</b>	<b>Name</b>	<b>Species</b>	<b>Movie</b>
1	Jumbo Jr.	Elephant	Dumbo
2	Ellie	Woolly Mammoth	Ice Age
3	Doug	Mastodon	One million BC

# Keys

- Keys are what determine the **identity** of a row
- Keys can be simple (single column) or **compound** (two or more columns)
  - Example: **(First Name, Last Name)**
  - This prevents two rows with the same combination of first and last name
- You can have **primary** and **alternate keys**
  - Usually a good idea to keep a primary numeric key as well as others you may want...

<b>id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Age</b>
1	Homer	Simpson	39
2	Marge	Simpson	39
3	Bart	Simpson	10
4	Homer	Thompson	39
5	Homer	Simpson	28

# Foreign Keys

- A **key** from one relation can be a **column** in another
  - This is called a **FOREIGN KEY** constraint
- This can be used to establish a link **between** the table in different tables/relations
- **This is not automatic:** must be included in the **schema definition!**

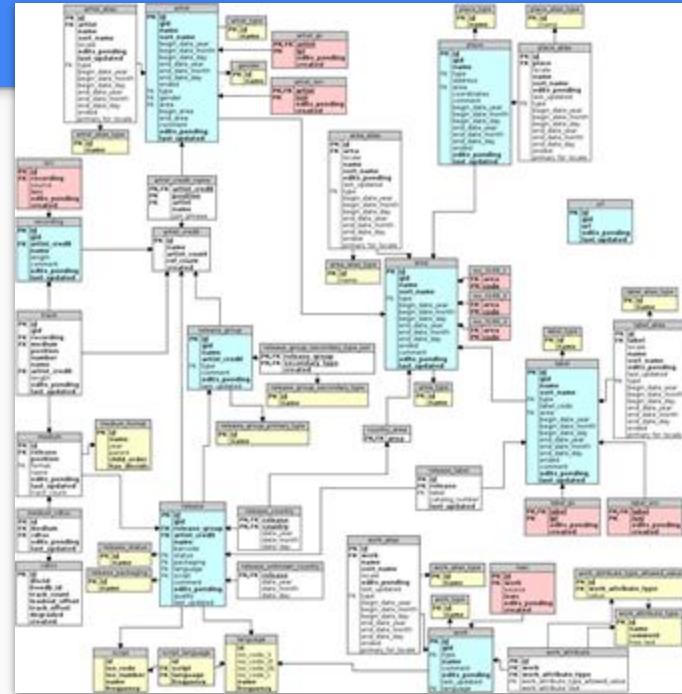
<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True



<b>id</b>	<b>Name</b>	<b>pachydermID</b>	<b>Movie</b>
1	Rocky	25	Jungle Book
2	Jumbo Jr.	1	Dumbo
3	Doug	3	One million BC

# Normalization

- A database schema is **normal** if data is not redundantly stored
  - Use **identifiers**, not **values**, to link between relations
- Modifying a record is easy if it exists in exactly one place
- But it can also be difficult
  - Reading complex data can be cumbersome
  - Multiple levels of indirection



[https://musicbrainz.org/doc/MusicBrainz\\_Database/Schema](https://musicbrainz.org/doc/MusicBrainz_Database/Schema)

# Summary

Relations are powerful!

- We use relational data every day without thinking of it
- Databases consist of one or more relations
- Putting data into a relational model can make it easier to work with
- Schemas provide some degree of safety and validation

# Structured Query Language (SQL)

- SQL is the language we use to talk to databases
  - Not a procedural language like Python or C
  - **Declarative**: state what you want, not how to compute it
- Think of it more like a **protocol** than a programming language
- SQL is an ANSI standard, but different implementations each have quirks
  - MySQL vs Postgres vs SQLite vs MSSQL vs Oracle SQL...

# SELECTing data

- Get all rows:

**SELECT \* FROM Pachyderms**

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False



<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

# SELECTing data

- Get all rows:  
Pachyderms

**SELECT \* FROM**

- Get some rows:  
Pachyderms

**SELECT \* FROM**

**WHERE Extinct = True**

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False



<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Woolly Mammoth	Pleistocene	Carnivore	True
2	Mastodon	Pliocene	Herbivore	True

# SELECTing data

- Get all rows:

```
SELECT * FROM Pachyderms
```

- Get some rows:

```
SELECT * FROM Pachyderms
```

```
WHERE Awesome =
```

True

- Get columns:

```
SELECT Era, Species  
FROM Pachyderms  
WHERE id > 2
```

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False



<b>Era</b>	<b>Species</b>
Pliocene	Mastodon
Miocene	Dataphant

# Selection

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True
4	Dataphant	Miocene	Data	False

- Remove tuples by filtering (WHERE ...)
- And remove / rename / reorder columns
- Result of SELECT is always another relation
- You typically iterate over rows produced by SELECT in your host language

```
for row in db.execute('SELECT * FROM Pachyderms):  
    print(row)
```

Python + sqlite3 example

# Joining relations

- Data is typically structured across multiple relations
- We can combine relations by **JOINing**
- **SELECT \*** from Pachyderms **JOIN** Character

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Extinct</b>
1	Elephant	Holocene	Herbivore	False
2	Woolly Mammoth	Pleistocene	Carnivore	True
3	Mastodon	Pliocene	Herbivore	True



<b>id</b>	<b>Name</b>	<b>Species</b>	<b>Movie</b>
1	Jumbo Jr.	Elephant	Dumbo
2	Ellie	Woolly Mammoth	Ice Age
3	Doug	Mastodon	One million BC

# A [?] JOIN B

Least specific



Most specific

<b>CROSS JOIN</b>	<b>All combinations</b> of rows ( $r_1, r_2$ ) $r_1 \in A, r_2 \in B \Rightarrow A \times B$ (no matching condition)
<b>[LEFT/RIGHT/FULL] OUTER JOIN</b>	<b>All rows are retained</b> from A (LEFT) or B (RIGHT), even if no match is found. Fill missing data with <b>NULL</b>
<b>INNER JOIN</b>	<b>Only matching rows</b> are retained (Like <b>OUTER</b> but without <b>NULLs</b> )
<b>NATURAL JOIN</b>	Rows must match on <b>all shared columns</b> (Special case of <b>INNER</b> )

# A [?] JOIN B

INNER and OUTER joins are most common

Least specific



Most specific

	<b>CROSS JOIN</b>	All combinations of rows ( $r_1, r_2$ ) $r_1 \in A, r_2 \in B \Rightarrow A \times B$ (no matching condition)
	<b>[LEFT/RIGHT/FULL] OUTER JOIN</b>	All rows are retained from A (LEFT) or B (RIGHT), even if no match is found. Fill missing data with NULL
	<b>INNER JOIN</b>	Only matching rows are retained (Like OUTER but without NULLs)
	<b>NATURAL JOIN</b>	Rows must match on all shared columns (Special case of INNER)

# Modifying data

```
INSERT INTO table (column1, column2, ...)  
VALUES (value1, value2, ...),  
      [(row_2_value1, row_2_value2, ...), ...]
```

```
UPDATE table  
SET column1 = value1, column2 = value2, ...  
WHERE [some condition]
```

# CAUTION:

## Use your RDBMS library to sanitize queries



```
db.execute("SELECT * FROM Pachyderms  
          WHERE species = '%s'" % name)
```

name variable becomes **part of** the query code!



```
db.execute(" SELECT * FROM Pachyderms  
           WHERE species = '?'", name)
```

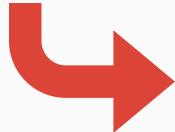
name variable is a **parameter** to the query code!

# Aggregation

# Aggregation queries

- Aggregation lets us summarize multiple tuples into a single result
- **Example:** find the average height of people within a zip code

**SELECT Zip, AVG(Height) FROM Residents GROUP BY Zip**



Zip	AVG(Height)
10003	2.68
10011	2

<b>id</b>	<b>Species</b>	<b>Height</b>	<b>Address</b>	<b>Zip</b>
1	Elephant	3.66	6 Washington Place	10003
2	Woolly Mammoth	2	60 Fifth Ave	10011
3	Mastodon	1.7	411 Lafayette St.	10003

# Some useful aggregators

- **AVG, SUM, MIN, MAX**  $\Leftarrow$  what you see is what you get
  - **COUNT(DISTINCT x)**  $\Leftarrow$  # of unique values of column x
  - **COUNT(\*) vs COUNT(x)**  $\Leftarrow$  # rows vs # non-nulls of a column
  - **GROUP\_CONCAT(x)**  $\Leftarrow$  concatenate (string) values  
**GROUP\_CONCAT(x, y)**  $\Leftarrow$  same, but join with string y

# Aggregation conditions

- **SELECT ... WHERE** [condition] **GROUP BY** [fields]
- **WHERE** clause applies to **input**, not **output**
- What if you only want to keep certain groups (e.g., sum > 10)?
  - ... **HAVING** [group condition]
  - **SELECT sum(Height) FROM TallPachys GROUP BY zip HAVING sum(Height) > 10**

# Indexing

# What is an index?

# Why do we (sometimes) use an index when using databases?

# Logical and Physical storage

- Relational schemas provide one view of the data
  - Set or list of **tuples**
- This may not be the best way to organize the data internally
  - Organizing by column can be much more efficient!
  - And what data structure do you use for each type? Hash tables? Trees?
- RDBMS abstract these decisions away from you (the user)
  - But sometimes you can help it out, if you know how data will be used

# Indexing

- An **index** is a data structure over one or more columns that can accelerate queries
- Example:
  - A table that has a few distinct values repeated millions of times
  - And you frequently want all rows with exactly one given value
  - It might be faster to store a mapping **value → rows** than to search each row independently

<b>id</b>	Name	Continent	Street	State
1	Elephant	Asia	Z1402 Krishi Bank Road	Matlab
2	Woolly Mammoth	Europe	Friedhofstr. 18	Ba-Wü
3	Mastodon	North America	60 Fifth Avenue	New York

# Drawbacks of indices

- They take time and **space** to construct
- **Composite indices** (multiple columns) are particularly costly
- **Updates** become slower
- No guarantee that they will help in all queries

# When to index?

- When data is **read more often** than written
- When queries are **predictable**
- When queries rely on a **small number of attributes**
- **Remember:** you can always add or delete indices later

# Summary

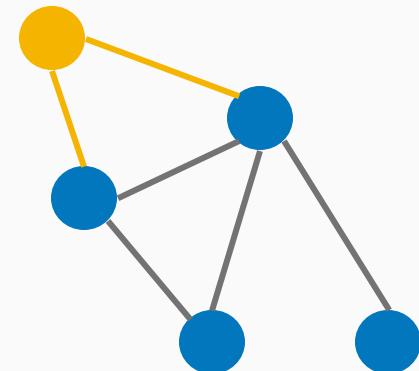
SQL is magic!

- SQL provides a standard interface to relational databases
- Modern frameworks often provide SQL-style interfaces
  - Pandas .groupBy(), .merge(), etc...
- Use indices to organize your data ahead of time

Databases (can) ensure the integrity of transactions

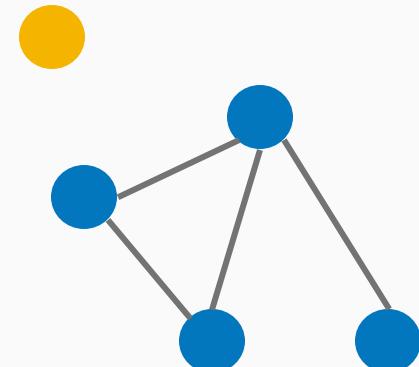
# More file system drawbacks: consistency!

- What if you want to impose **constraints** on your data?
- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?



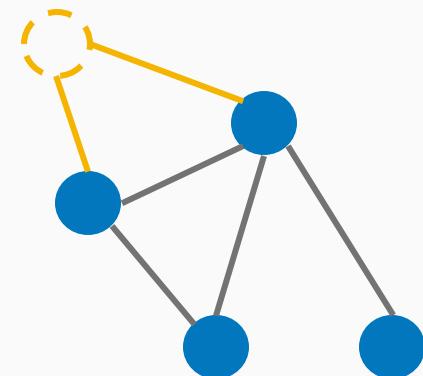
# More file system drawbacks: consistency!

- What if you want to impose **constraints** on your data?
- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?
- Add vertex first: graph is **disconnected**



# More file system drawbacks: consistency!

- What if you want to impose **constraints** on your data?
- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?
- Add edges first: **edges** are **invalid**



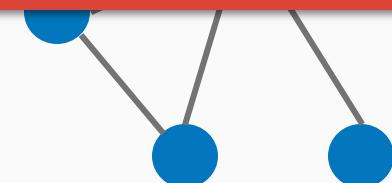
# More file system drawbacks: consistency!

- What if you want to impose consistency?
- Example:
  - Guaranteeing that a graph is connected
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?
- Add edges first: **edges** are **invalid**

Either operation by itself can render the data **inconsistent**.

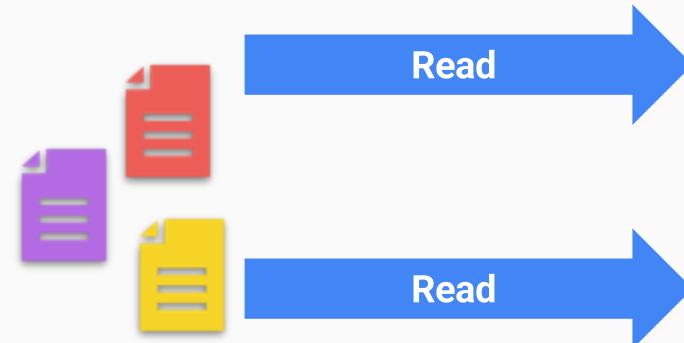
Operations need to be performed simultaneously, but file systems do not generally provide this functionality.

We need another layer of abstraction.



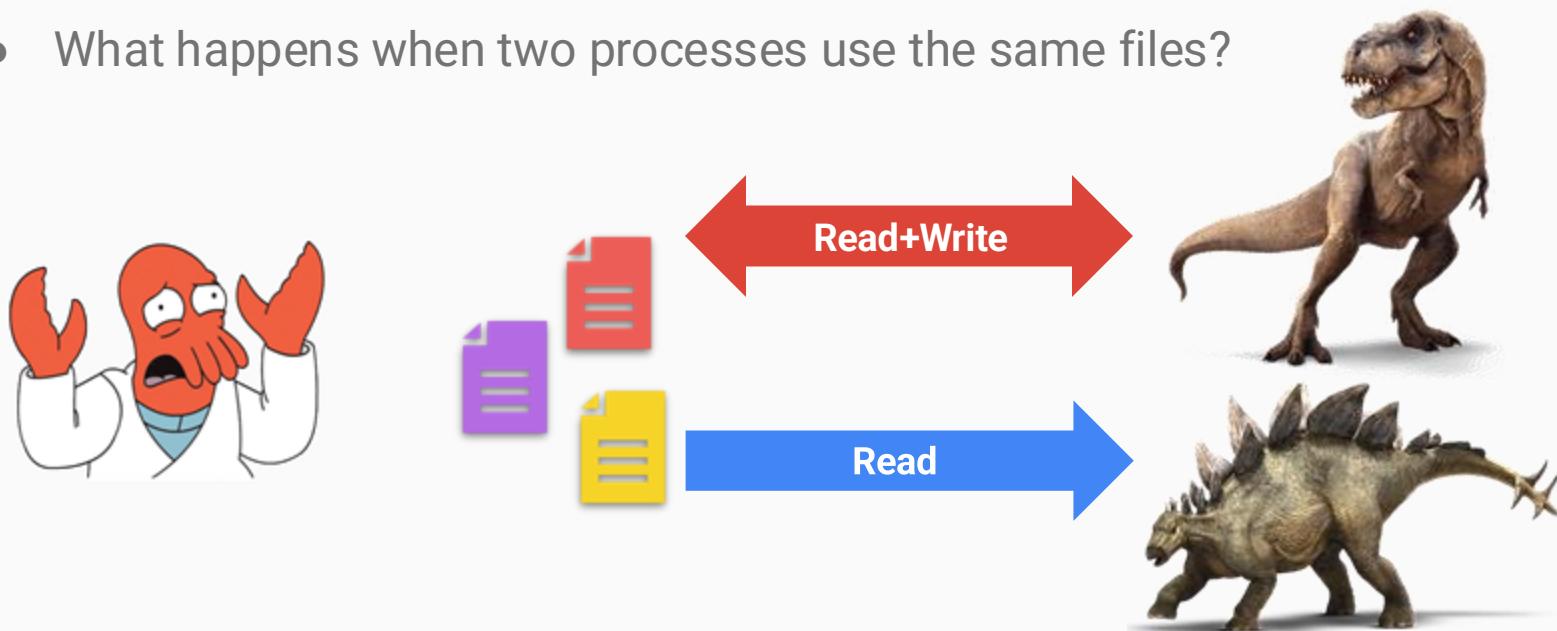
# More drawbacks: concurrency!

- What happens when two processes use the same files?



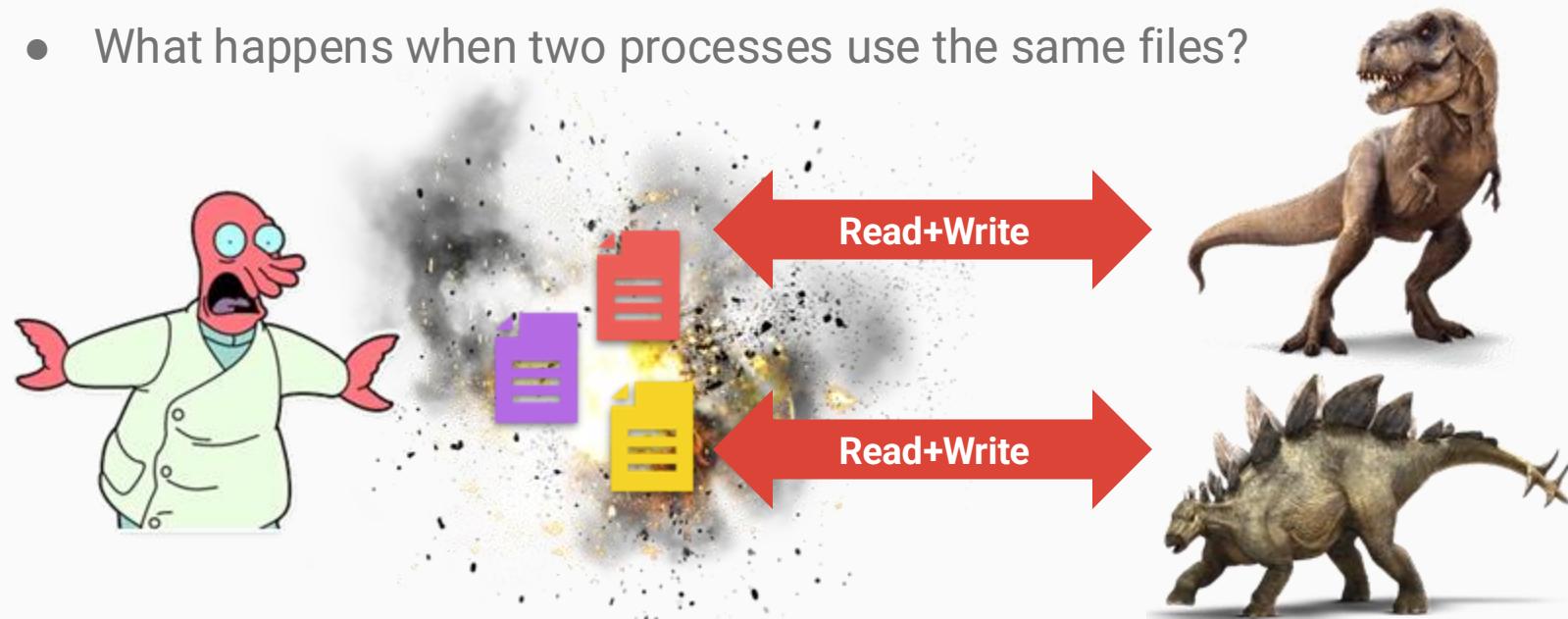
# More drawbacks: concurrency!

- What happens when two processes use the same files?



# More drawbacks: concurrency!

- What happens when two processes use the same files?



# ACID principles to the rescue

Atomicity	Operations are all-or-nothing (No partial updates; operations bundled in <b>transactions</b> )
Consistency	Transactions move from one <b>valid</b> state to another (only!)
Isolation	Concurrent operations do not depend on <b>order</b> of execution
Durability	Completed transactions are <b>permanent</b> (usually implemented by flushing to disk before completion)

# Atomicity in practice

- When modifying tables, wrap query statements in  
**BEGIN TRANSACTION**; [queries]; **COMMIT**;  
or  
**BEGIN TRANSACTION**; [queries]; **ROLLBACK**;
- Different DBMS use slightly different syntax (**START**, **BEGIN**)
- If a query **fails** mid-transaction, uncommitted changes will be **abandoned**
  - ⇒ DB is always left in a **valid state**

# Aside: transactions example

- SQL transactions are kind of like “try ... except” blocks in Python/Java/etc.
- It’s helpful to think of SQL interactions as a conversation or **session**
  - Each command executes and either succeeds or fails
- If you’re performing multiple queries, and one of them fails for some reason, you can **roll back** to the state of the database at the beginning
- If everything completes successfully, you can **commit** the transaction

# The classic example

id	balance ( $\geq 0$ )
1	\$10
2	\$20

- Imagine transferring \$20 from account id=2 to account id=1
  - “balance” column cannot go negative
- This is done with two update queries in a transaction:
  - BEGIN TRANSACTION
    - UPDATE account SET balance=balance-20 WHERE id=2
    - UPDATE account SET balance=balance+20 WHERE id=1
  - COMMIT
- These need to either both happen or neither happens

# The classic example

id	balance ( $\geq 0$ )
1	\$10
2	\$20

- If we instead wanted to transfer \$20 from id=1 to id=2, this would violate the schema constraint “balance  $\geq 0$ ”.
- This is done with two update queries:
  - BEGIN TRANSACTION
    - UPDATE account SET balance=balance-20 WHERE id=1 ← fails!
    - UPDATE account SET balance=balance+20 WHERE id=2
  - ROLLBACK
- So the second query is not committed - all changes are reverted!

# Consistency in practice

- **Consistency** is maintained by **schema**
- Schema can add basic value checks as well

```
CREATE TABLE Pachyderms (id INTEGER PRIMARY KEY,  
                         species TEXT NOT NULL,  
                         height NUMBER          CHECK (height >= 1.0))
```

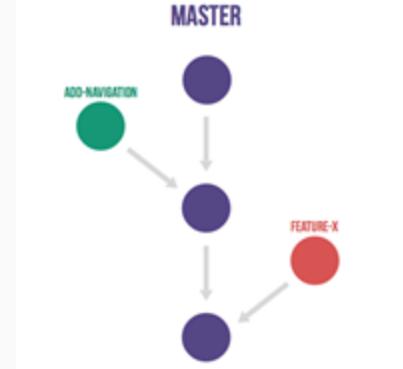
- If data does not fit the **schema**, the operation **fails immediately**
- ⇒ DB cannot enter an **invalid state**

# Isolation in practice

- Usually achieved by **locking** the database during modification
  - Only one transaction can hold the lock at a time
- This becomes a real problem for distributed databases!
  - Locking the entire DB would stall everyone
- As we'll see next week, **Map-Reduce** side-steps this problem

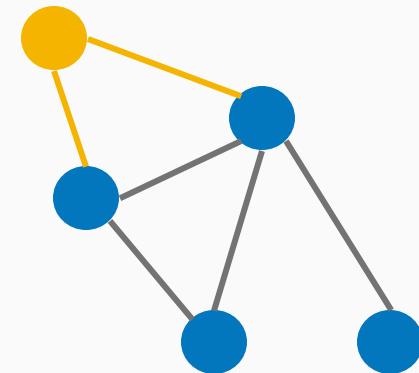
# You've seen some of this already (In IDS – if not, you saw it in the lab last week)!

- **git** can be seen as a kind of distributed (non-relational) database
- **Atomicity:**
  - Changes are staged independently (**git add**) into a transaction
  - Transactions are finalized atomically (**git commit**)
- **Consistency:**
  - Conflicting changes are detected and forbidden (**merge conflict**)
- **Isolation:**
  - Different **branches** or **clones** can be modified independently
- **Durability:**
  - Objects are saved to the local repository



# Revisiting the connected graph problem with properties of reliable transaction processing (ACID) in mind:

- What if you want to impose **constraints** on your data?
- Example:
  - Guaranteeing that a graph is **connected**
  - Vertices stored in **nodes.dat**
  - Edges stored in **edges.dat**
  - What if you want to add a **vertex** and two **edges**?



Which ACID property would be most useful for fixing our connected graph problem and why?

# Summary

Relational database  
management systems

- The relational model is powerful!
  - Tables and joins are a simple, flexible model for many kinds of data!
  - SQL is a little strange, but powerful
- RDBMS provide
  - Data abstraction
  - A common language to interfacing with data (SQL)
  - Concurrent access

# Next week

- Distributed computation with Map-Reduce
- Reading:
  - First: Dean & Ghemawat
  - Second: DeWitt & Stonebraker

**Q & R**



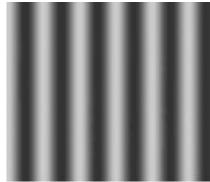
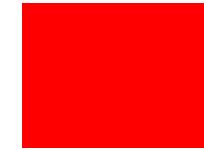
Smallest font



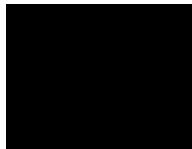
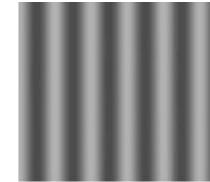
Please turn off and put  
away your cell phone



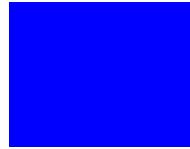
# Calibration slide



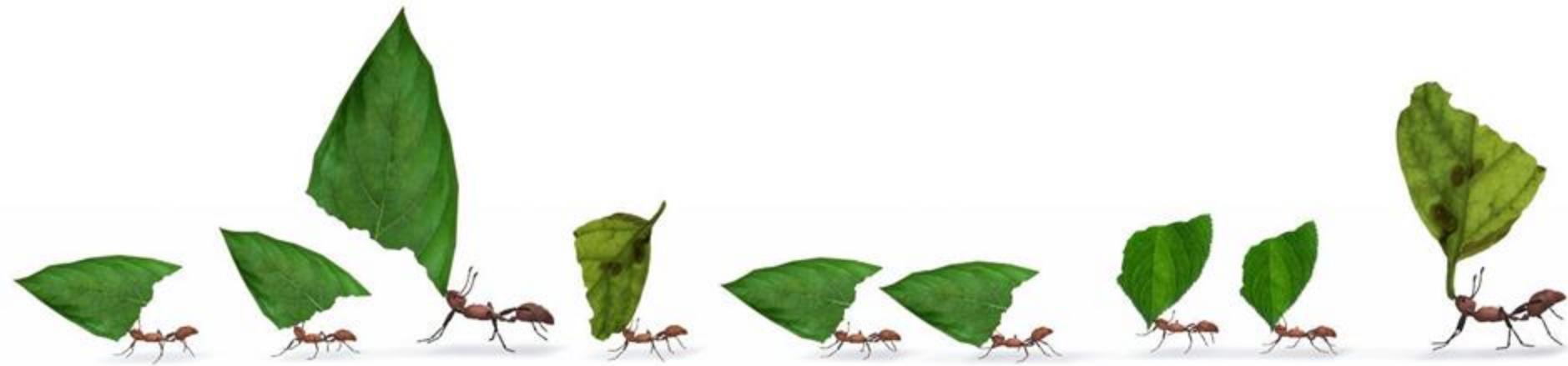
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



# Week 03: Map Reduce

DS-GA 1004: Big Data



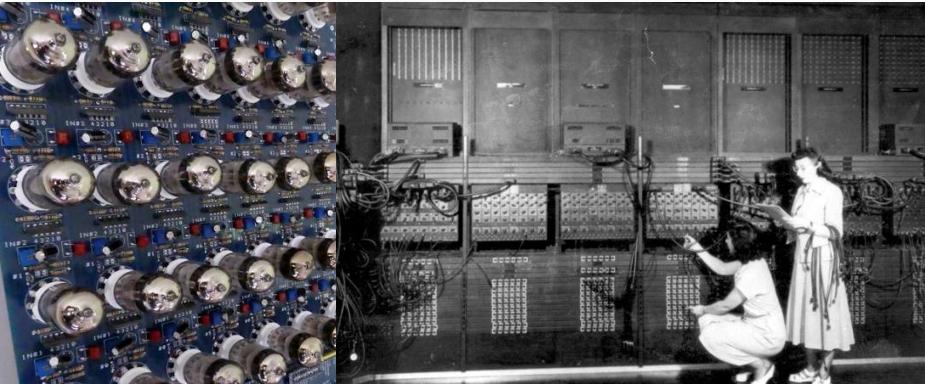
# Announcements

- Everyone got an HPC account now
- This week: Lab 3 (MapReduce)
- HW 1 is due tomorrow (02/11)
- Quiz in lab this week
- HW2 releases this week (due 02/27)

A brief history of how we store,  
manage, access data

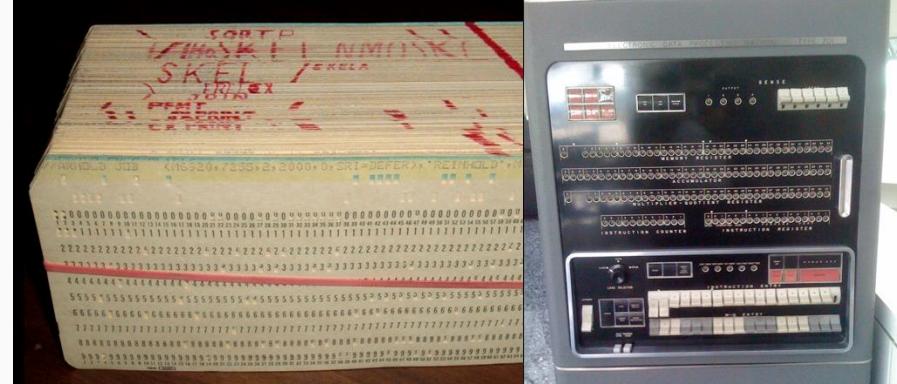
# “Prehistory” (not covered in this class)

1940s



Vacuum tubes & plugboards  
All hardware

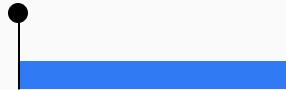
1950s



Punchcards  
“Software” (programs)

## **File systems**

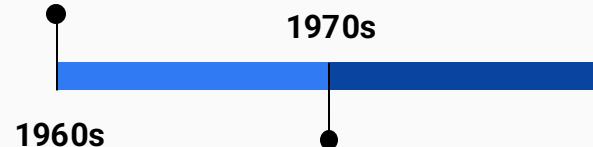
Custom software for  
each application / query



**1960s**

## File systems

Custom software for  
each application / query



**Relational model  
(Codd, 1970)**

Tables are good!

Structured Query  
Language (SQL)

## File systems

Custom software for each application / query

## RDBMS takes off

Databases for commodity computers

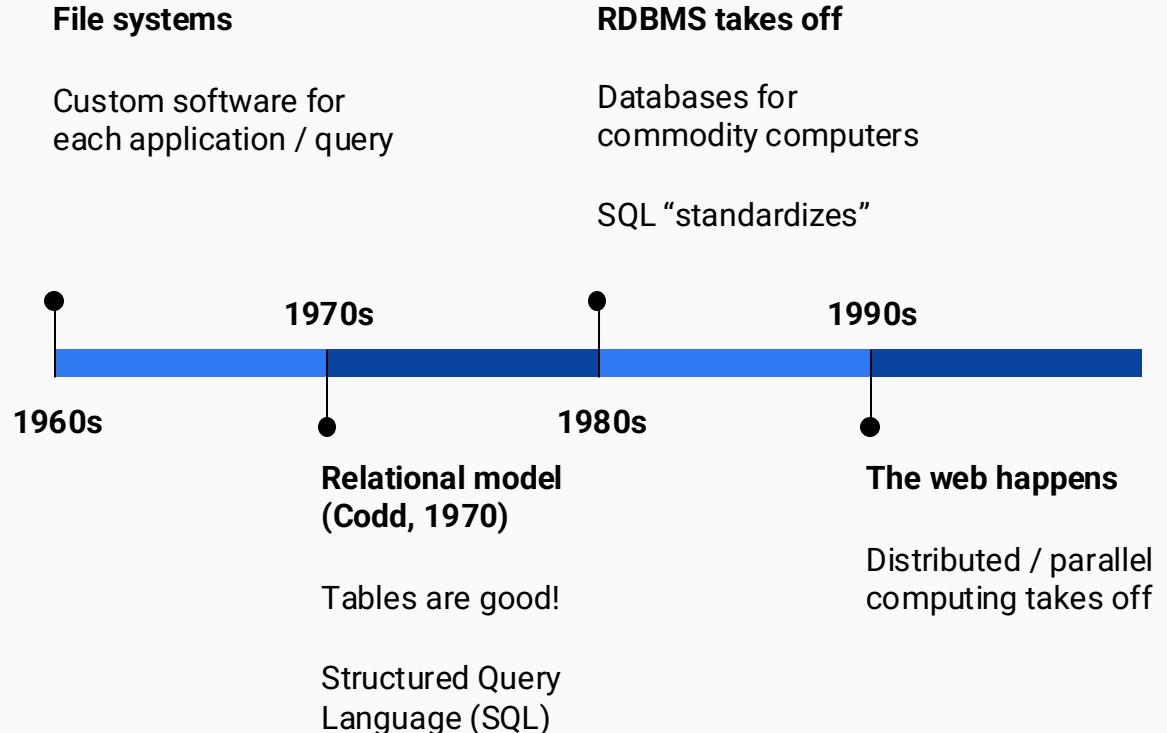
SQL “standardizes”

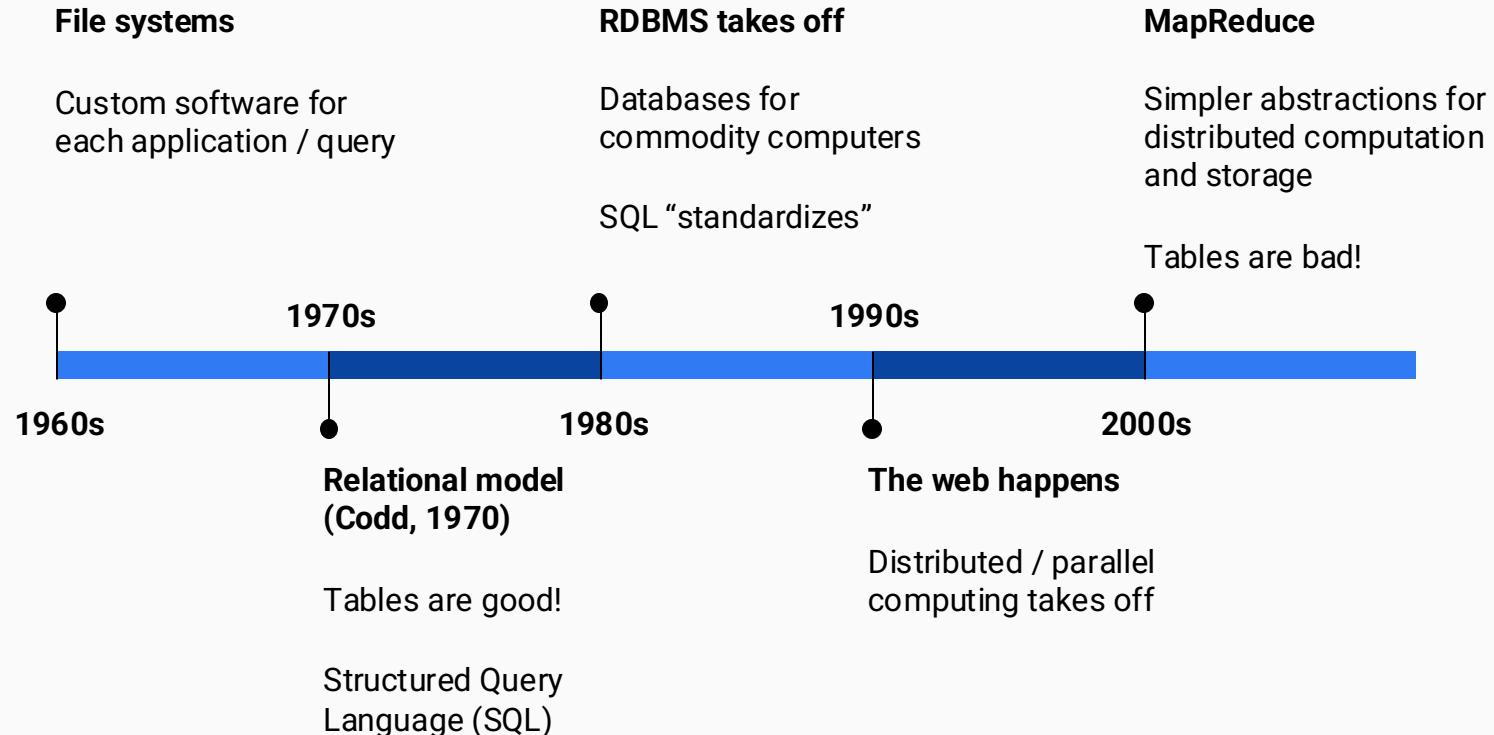


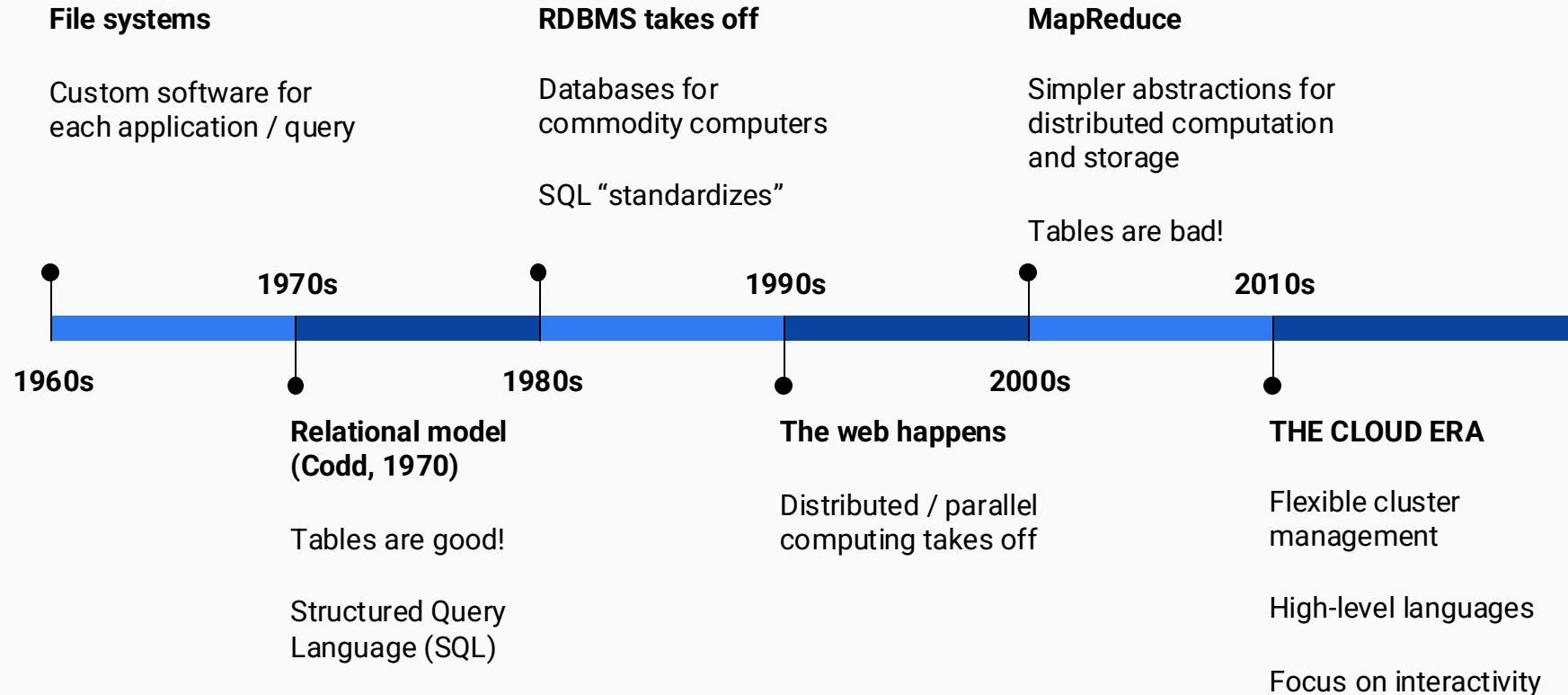
**Relational model  
(Codd, 1970)**

Tables are good!

Structured Query Language (SQL)







Tables are good if you call them 🦩 data frames 🦩

# Reminder: Last time - RDBMS

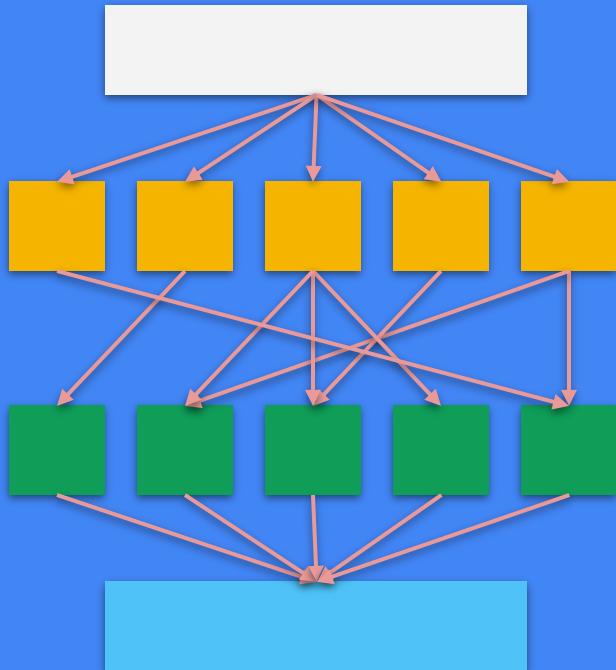
- **relations and schemas** standardize the shape of data
- **SQL** standardizes data interactions
- **DBMS** hides the implementation details
- **Transactions** provide safety and concurrency (ACID principles)

<b>id</b>	<b>Species</b>	Era	Diet	Popular
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False



<b>id</b>	Name	<b>Species</b>	Internals
1	Earl Sinclair	Megalosaurus	Puppet
2	Grimlock	T. Rex	Robot
3	Snarl	Stegosaurus	Robot

# This week



1. Resolving confusions / doubts / struggles from AAA
2. Introduction to Map-Reduce  
(Dean & Ghemawat, 2008)
3. Assessment of Map-Reduce  
(DeWitt & Stonebreaker, 2008)

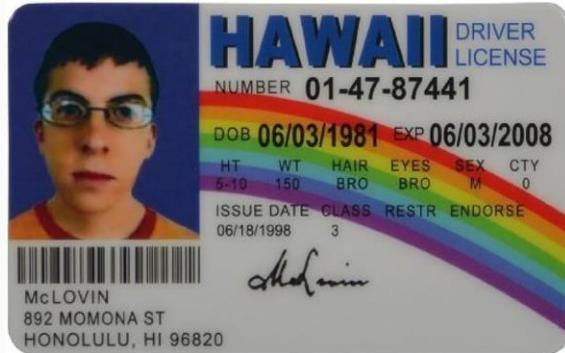
# Confusion, Doubt, & Struggle: Keys

“Keys” are a central concept in many CS applications

We have already seen the use of keys in SQL last week,  
will see keys used today in MapReduce  
and will see the use of keys in the future

If you do not have a CS background, you  
might have an unhelpful association:

Unless specified  
otherwise, when  
you see “key” in CS,  
think unique “ID”:



# Confusion, Doubt, & Struggle: Normalization (1NF)

Introduced by Cobb (1970): 5 normal forms – 1NF, 2NF, 3NF, 4NF, 5NF

Philosophy: Normalized databases avoid needless duplication of information (= redundancy)

Purpose: Normalized databases are easier to maintain and also take up less space

For instance: Every row has to be unique (to comply with 1NF)

Name	Major
Alex	Data Science
Brett	Computer Science
Corey	Engineering
Drew	Business
Emory	Mathematics
Alex	Data Science



**Not normalized**

**(Primary) key**

N-number	Name	Major
18994092	Alex	Data Science
18994093	Brett	Computer Science
18994094	Corey	Engineering
18994095	Drew	Business
18994096	Emory	Mathematics
18994097	Alex	Data Science

**Normalized**

# Confusion, Doubt, & Struggle: Normalization (beyond 1NF)

The normal forms build on each other. So once it is 1NF compliant: Does it make higher NF?

All non-primary key information should fully depend on the primary key, no transitive dependencies

Heuristic: Tables should contain information about a single entity or concept and relate to each other.

N-number	Name	Major	GPA	Gender	Phone number	Tuition
18994092	Alex	Data Science	3.5	M	998-3307	\$70,000
18994093	Brett	Computer Science	3.7	F	555-5555	\$50,000
18994094	Corey	Engineering	3.8	M	998-1212	\$60,000
18994095	Drew	Business	3.3	F	123-4567	\$85,000
18994096	Emory	Mathematics	2.9	M	998-7920	\$45,000
18994097	Alex	Data Science	3.9	F	212-0000	\$70,000

## Normalized?

# Confusion, Doubt, & Struggle: Normalization (beyond 1NF)

A normalized version of the database from the last slide creates 3 tables that relate to each other:

**Students table**

N-number	Name	Gender	Phone #
18994092	Alex	M	998-3307
18994093	Brett	F	555-5555
18994094	Corey	M	998-1212
18994095	Drew	F	123-4567
18994096	Emory	M	998-7920
18994097	Alex	F	212-0000

**Majors table**

mID	Major	Tuition
1	Data Science	\$70,000
2	Computer Science	\$50,000
3	Engineering	\$60,000
4	Business	\$85,000
5	Mathematics	\$45,000

**Academic records table**

N-number	mID	GPA
18994092	1	3.5
18994093	2	3.7
18994094	3	3.8
18994095	4	3.3
18994096	5	2.9
18994097	1	3.9

**Normalized?**

Why create 3 tables? Why not put the tuition in the academic records table?

Heuristic for this class: When in doubt, make a new table, that uses the keys of other tables

# Confusion, Doubt, & Struggle: Index

- Building an index is a core CS topic.
- Given the scale of the data we're working with, the need for speed is always a consideration.
- At first pass, an index makes things faster.
- An index makes things faster because it – specifically – speeds up search, allowing to find relevant information faster.
- An index is a data structure that has to be built.
- It is effectively a lookup table, just like the index in a book (that's the analogy – why it is called an index in the first place, same reason)
- An index still has to be searched, but it is usually much faster to search the index instead of all (the rows of) the data.
- So an index provides a shortcut that narrows down our search space.
- It tells us where to jump to, which is particularly helpful if all similar/relevant data is located in the same neighborhood.
- There are many implementations, for instance clustered index vs. non-clustered index.
- Sample Query:
- CREATE INDEX idx\_column\_name ON table\_name(column\_name);

## J

JAVA language, [9](#)

JAVASCRIPT language, [9](#)

JULIA language, [9](#)

## K

Kaiser window, [146](#), [146f](#), [149–150](#)

Kernel, [119–120](#)

## L

L1 regularization, [214](#)

L2 regularization, [214](#)

Lasso methods, [214](#)

Lasso regressions, See [L1 regularization](#)

Latency to first spike, [53](#), [53](#), [55](#), [79–80](#)

LDA, See [Linear discriminant analysis \(LDA\)](#)

Leak conductance, [162](#)

leaky\_integrate\_and\_fire method, [167](#)

len function, [63–64](#)

LFPs, See [Local field potentials \(LFPs\)](#)

Linear discriminant analysis (LDA), [225](#)

Linear regression, [195](#), [195](#)

Linearization process, [32](#), [34](#)

linearizedSpikeTimes cell array, [101](#), [102](#)

list comprehension, [65](#)

NOW

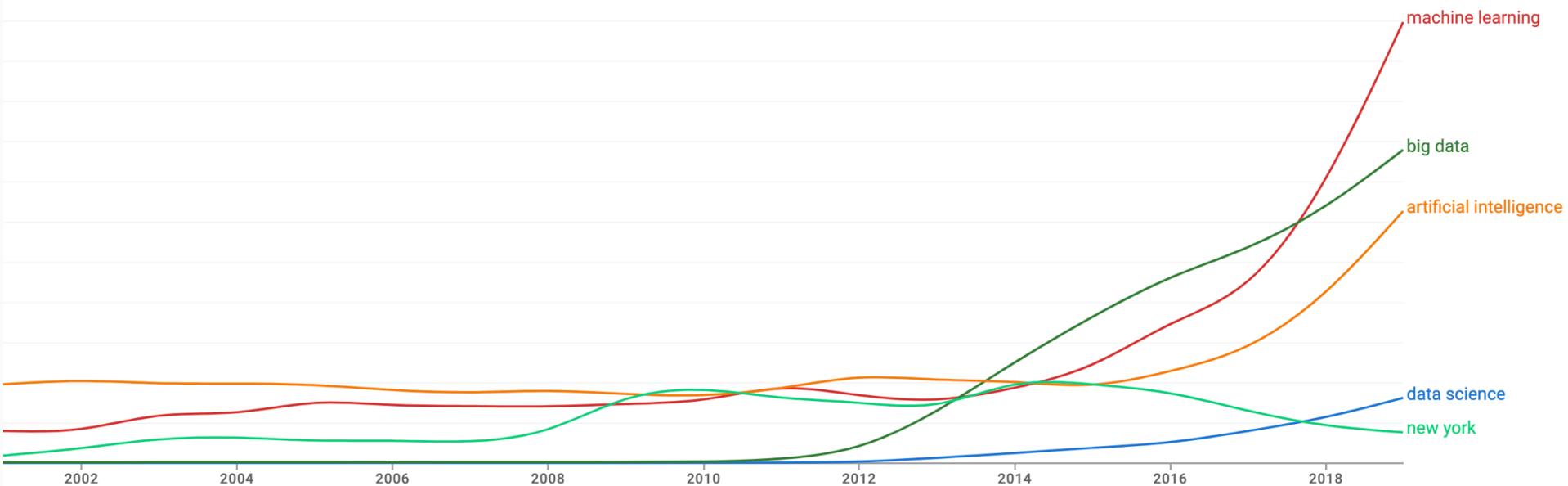
# *MapReduce mapreduce*

# *Map-Reduce map reduce*

I have a working knowledge of Hadoop (defined as using it regularly on the job or other activities, e.g. research) to the point where I would claim to be "proficient" or "fluent" in it on a resume, and could answer most questions that might come up during a technical interview without looking anything up



# A typical use case: Google's Ngram Viewer

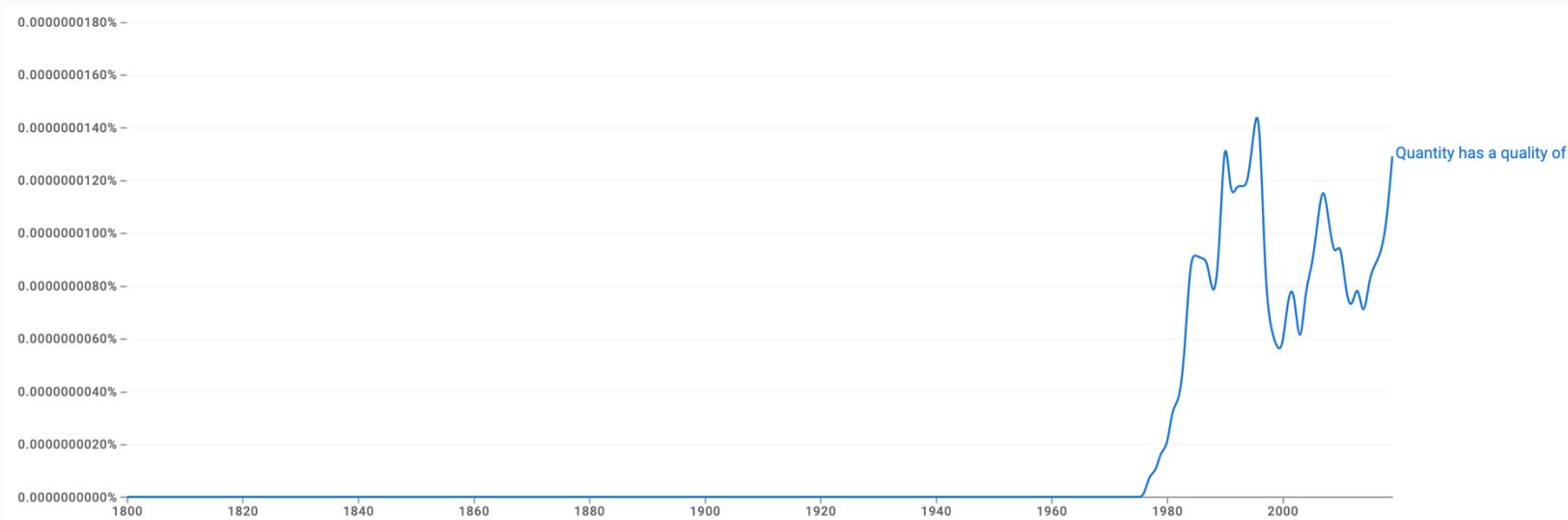


Challenge: A tremendous **collection of documents** needs to be searched to do something like this

Is a relational database a good way to implement such an application?

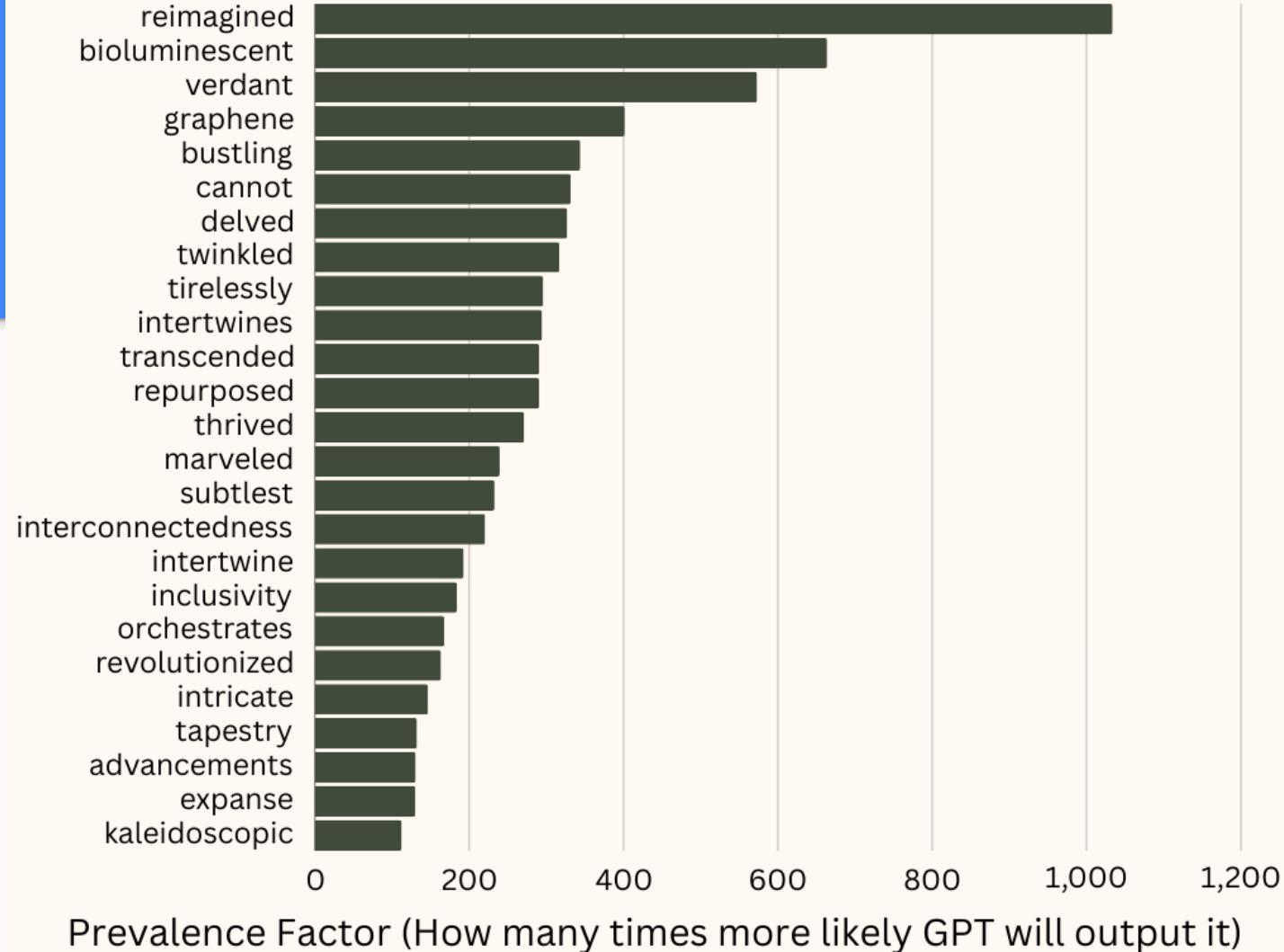
# Why Map Reduce was created

- All of the algorithmic steps to implement word counts are simple/straightforward
- Where does the complexity come from?
- “Quantity has a quality of its own” (Thomas Callaghan, 1979)



- All of the complexity comes from doing this task at overwhelming scale.
- Map Reduce provides a standard, usable way to handle this complexity.

# Word frequency analysis can reveal very important information



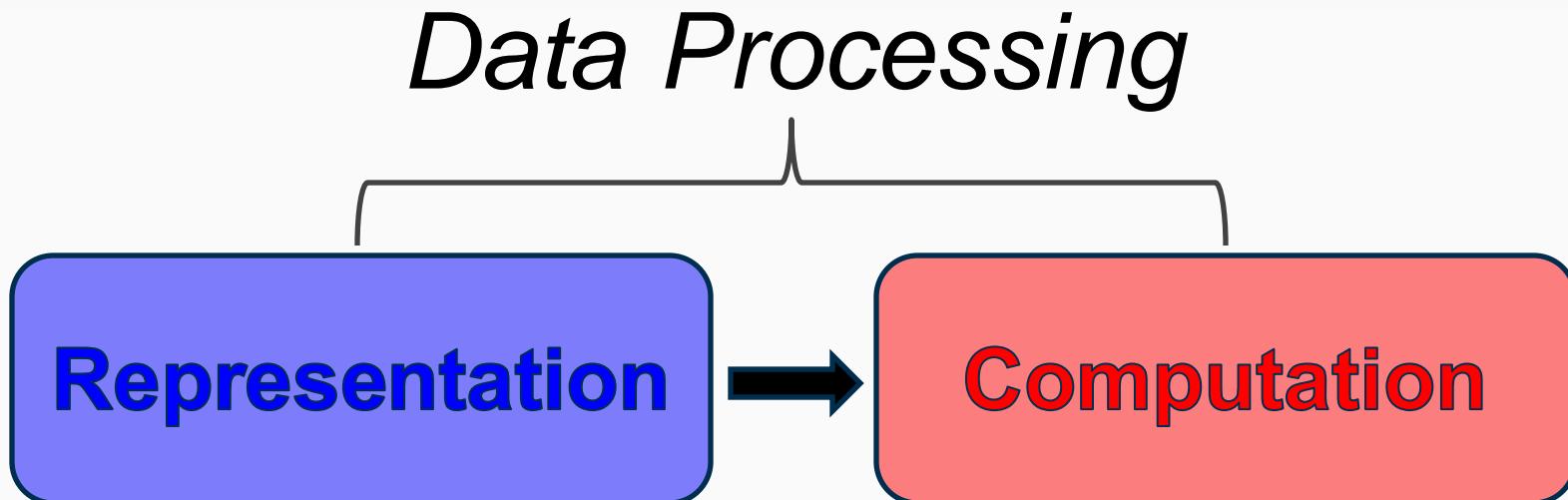
# Google's Motivation: text indexing

- Say you have  $N$  documents (with  $N$  very large, e.g. the web), and you want to construct an index: **words** → **documents**
- On a single machine, this process takes  $\Omega(N)$  time
- **Observation:** this problem is (almost) embarrassingly parallel
  - Whether any word appears in a document is **independent** of other documents
  - We should be able to process documents independently and **combine the results**

# Text indexing continued

- You could have multiple computers write to a shared database
  - With  $M$  machines, can we lower the time to  $\Omega(N/M)$ ? ← **Goal**
- You need to find some way to **distribute** work (data?) and **aggregate** results
- **Map-Reduce** (Dean & Ghemawat, 2004) provides a framework for this
- **Hadoop** (2008-) provides an open source implementation of Map-Reduce
  - ... and supporting infrastructure for distributed computing

# The philosophy of MapReduce



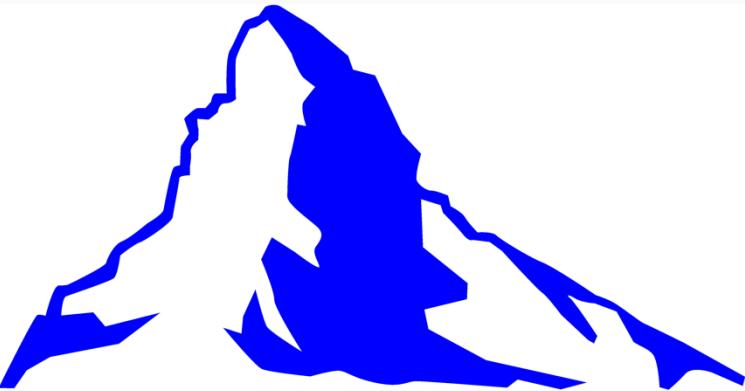
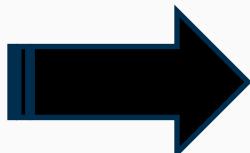
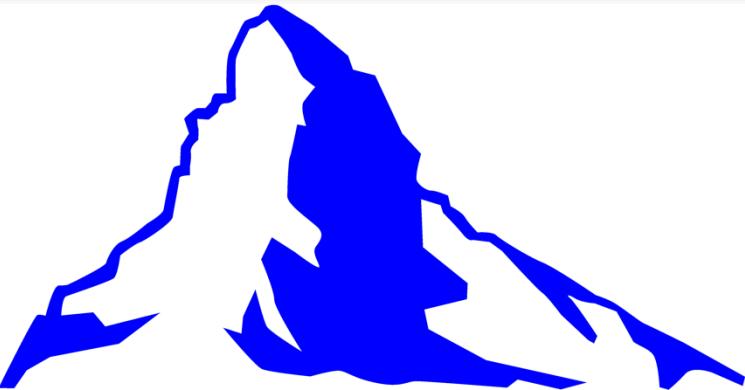
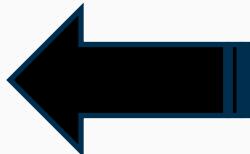
Computations have to consist of a “map” and a “reduce” step (only)

# Why do such a thing?

# “Data Locality”

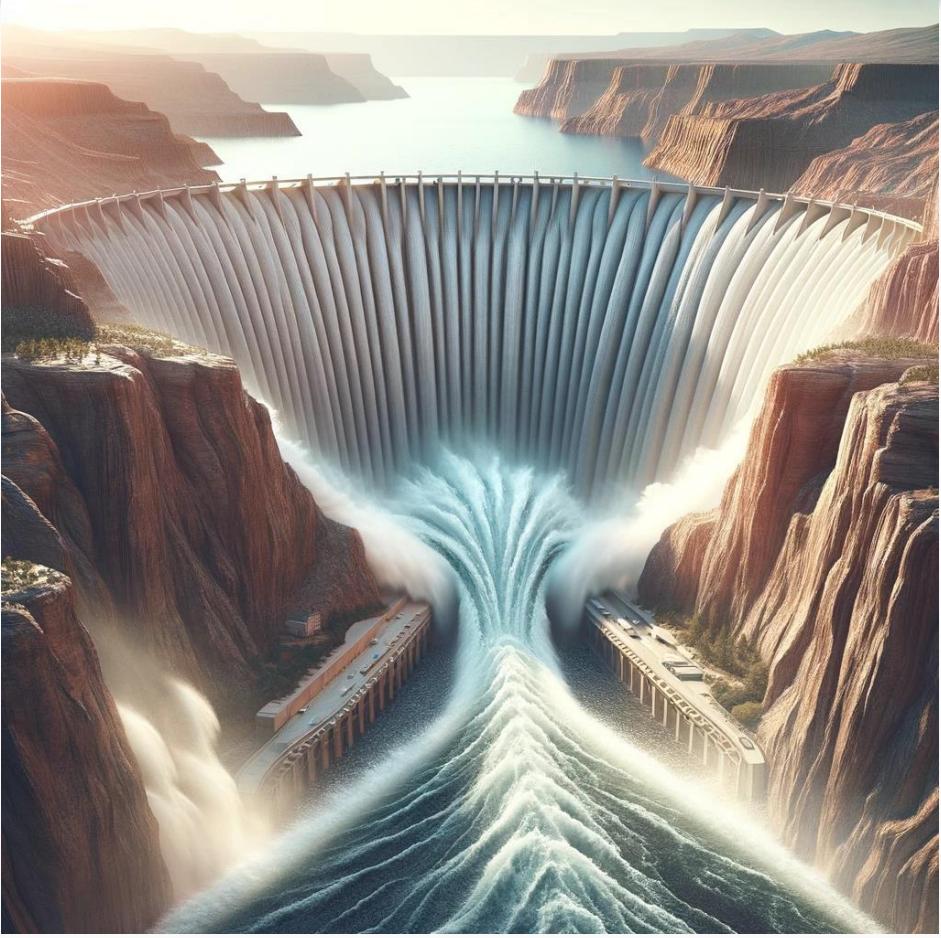
Computation

Data



No communication!

# Power through restrictions and constraints



This is going to be a general theme in this class / field

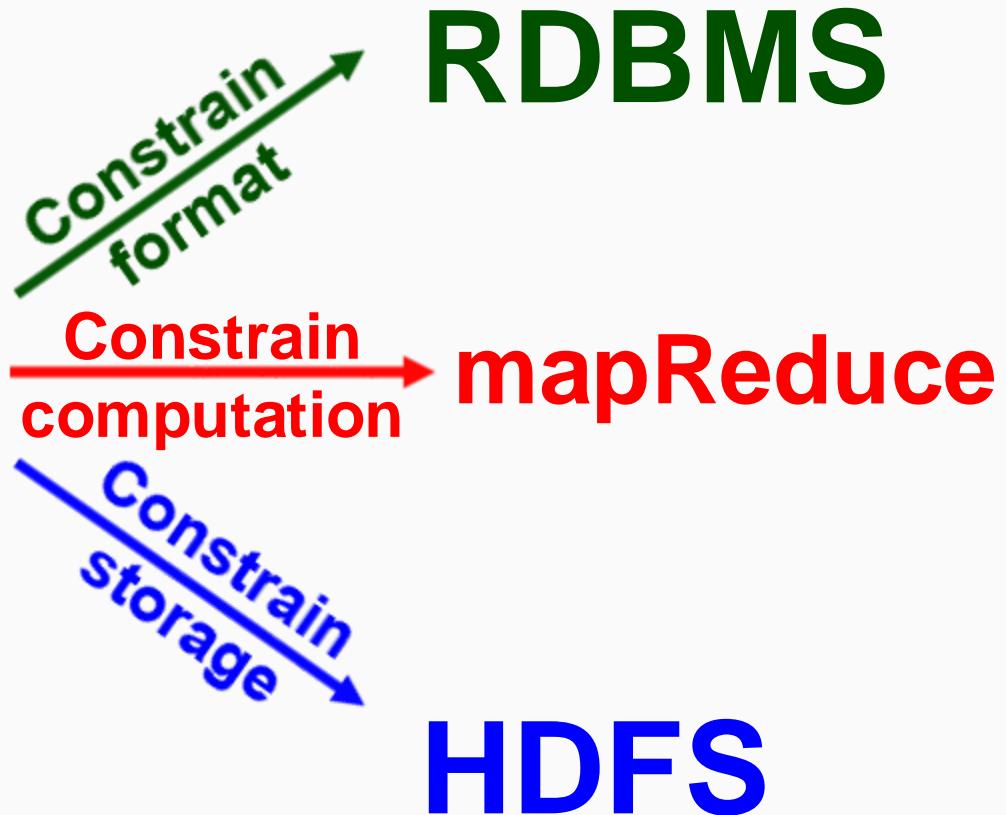
# File systems

Flexibility maxing:

Content / Format can be anything

Any computation can be applied

Storage can be modified in any way



# Why map-reduce?

- Distributed programming is really **difficult!**
- If we **restrict** how we program, parallelism becomes easier
- The **map** and **reduce** operations are surprisingly powerful!

# Why “map” and “reduce”, specifically?

- These are common operations in **functional programming**
  - E.g.: LISP, Haskell, Scala...
- **map(function  $f$ , values  $[x_1, x_2, \dots, x_n]$ )  $\rightarrow [f(x_1), f(x_2), \dots, f(x_n)]$** 
  - **map** : function, list  $\rightarrow$  list
  - $f$  is applied element-wise
  - This allows for parallelism, if list elements are independent
- **reduce(function  $g$ , values  $[x_1, x_2, \dots, x_n]$ )  $\rightarrow g(x_1, \text{reduce}(g, [x_2, \dots, x_n]))$** 
  - **reduce** : function, list  $\rightarrow$  item
  - $g$  is applied recursively to pairs of items

# MapReduce: Conceptual framework

- You (the programmer) provide two functions of this form: **mapper** and **reducer**
  - These can be arbitrarily complex, but **simpler is better!**
- The **mapper** consumes inputs, produces/"emits" outputs of the form:  
**(key, value)**
- The **reducer** consumes a single **key** and list of **values**, and produces **values**
  - Reducer is **not** applied recursively like in functional programming
  - "reducer" is just a suggestive name borrowed from this framework (as an analogy)

# Map-Reduce flow

## 1. Map phase

- Distribute data to mappers
- Generate intermediate results (*key, value*)

## 2. Sort / shuffle phase

- Assign intermediate results to reducers (by *key*)
- Move data from mappers to reducers

## 3. Reduce phase

- Execute reducers and collect output

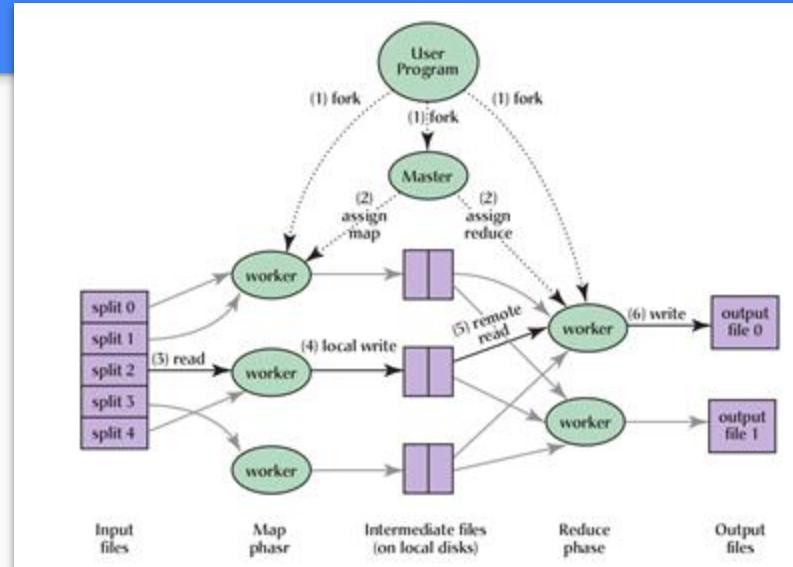
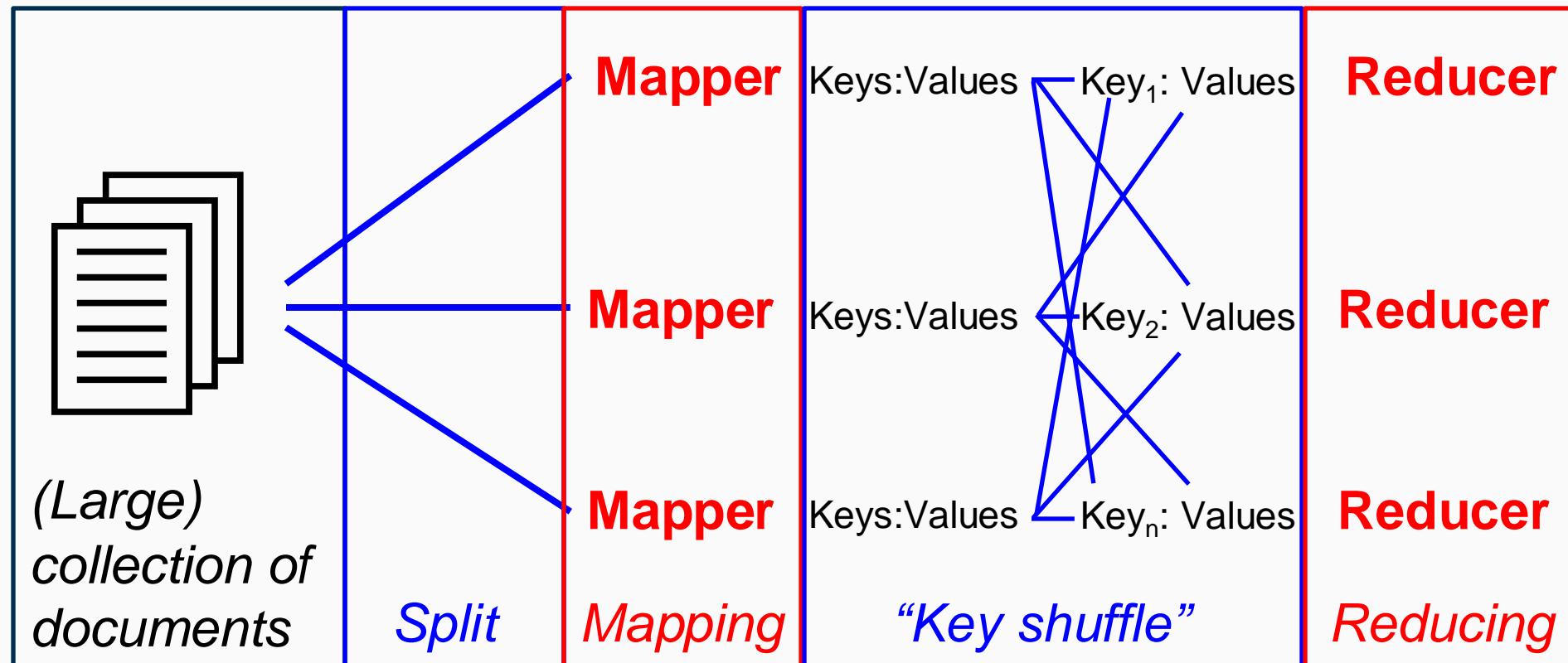


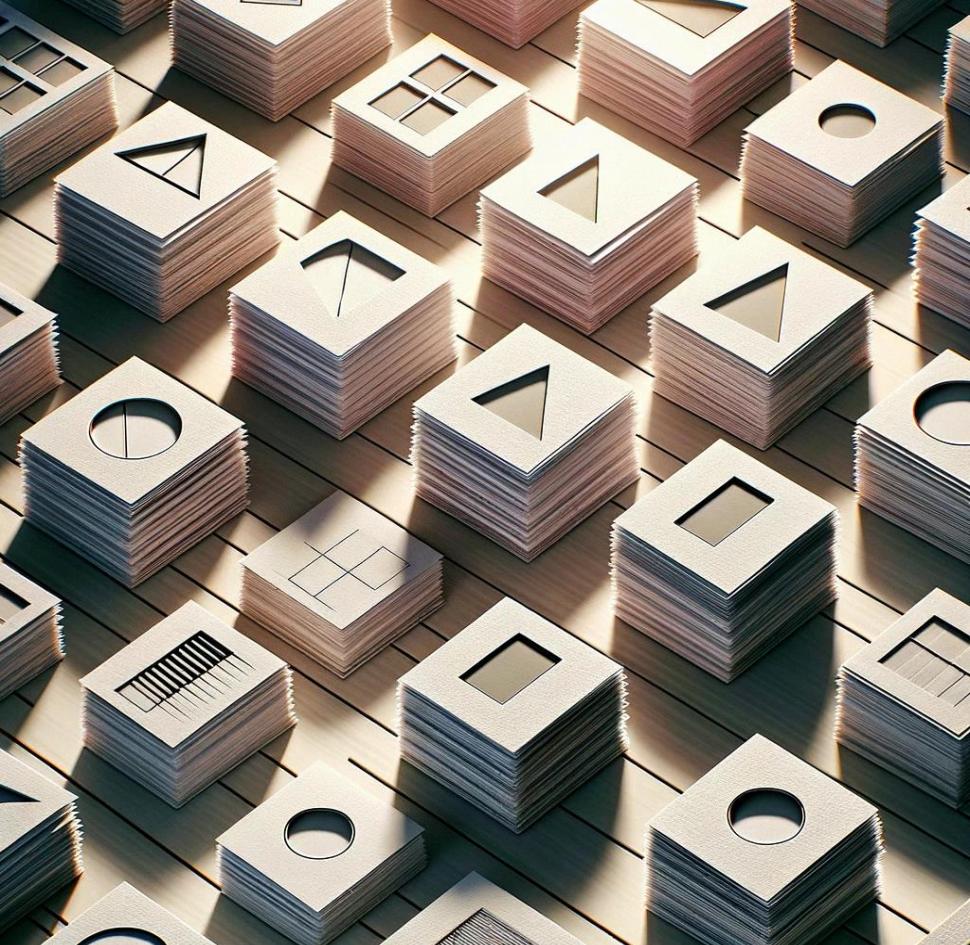
Figure from Dean & Ghemawat, 2008

# The MapReduce algorithm (essential schematic)

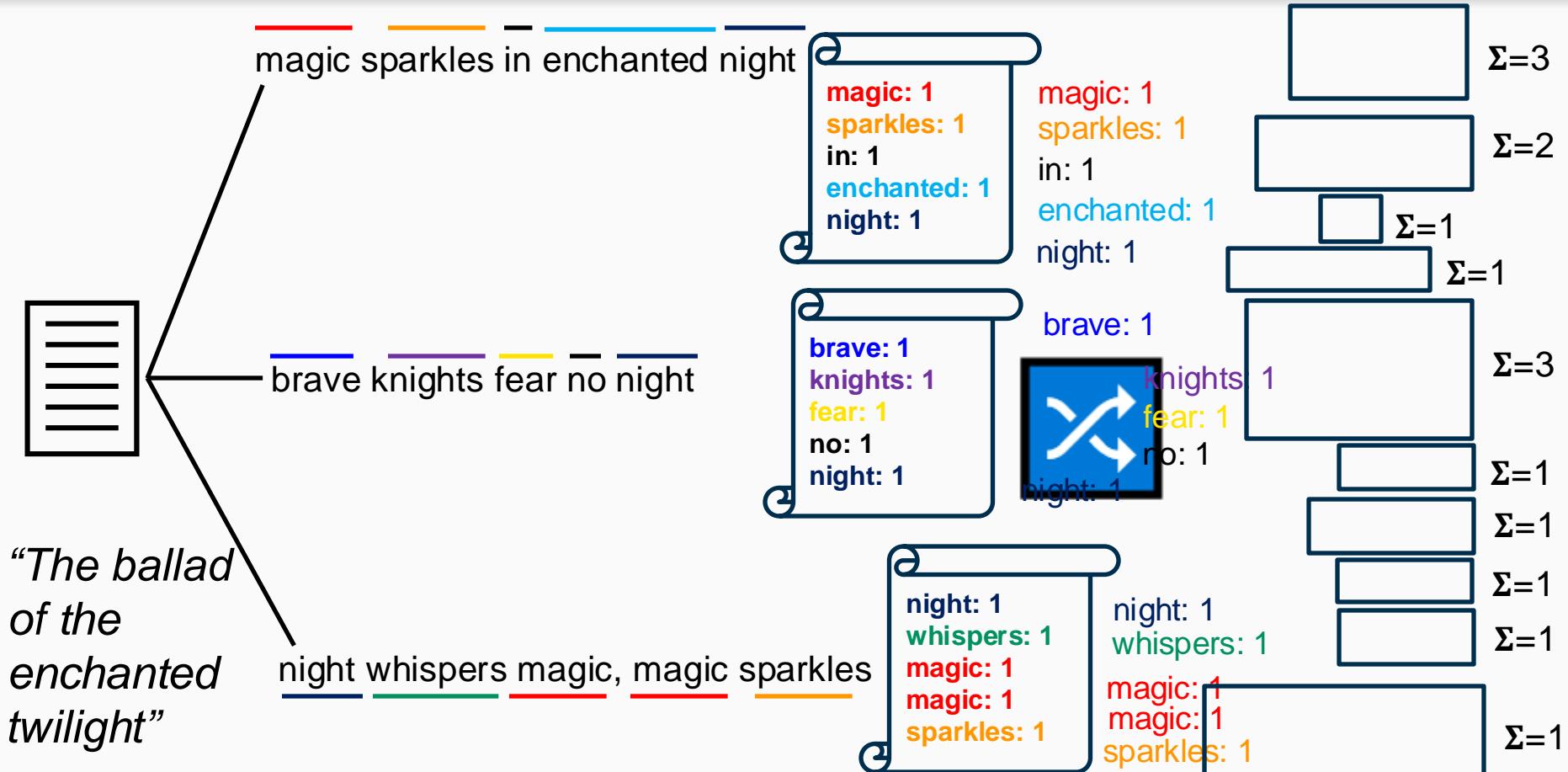
Little movement of **data** and no **communication** between **computational** units  
= massively **scalable** due to **parallelization** → “finish in reasonable time”



# Potential for confusion: Key shuffle vs. grouping by identity?



# The classic use case: Word counts



Again: Why do we restrict ourselves to “mappers” and “reducers” only?

It implements the philosophy / framework of **DEI**

**DEI**: “**Divide Et Impera**” (Divide and rule, since ~90 BC)

The **map** function is “embarrassingly parallel” – it is applied to the piece of the (divided) input data independent of others (also **stateless**)  
→ No need for communication between mappers.

The **reduce** function is an aggregator and relies on associativity:

**Associativity**:  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$  – order of operations matters not

This ensures that partial results can be merged in any order without affecting the final outcome (critical as order is not guaranteed).

**Determinism**: Same inputs yield same outputs, without side effects.

**Scalability**: If all true → easily achieved by adding processing nodes

# Just CS use cases or are there also DS ones?



A large  
collection of  
sales  
receipts



What would the **splitting** stage look like here?

What would the **mapping** stage look like here?

What would the **keyShuffle** stage look like here?

What would the **reducer** stage look like here?

# Working with Map-Reduce: Practical considerations

# Some tips...

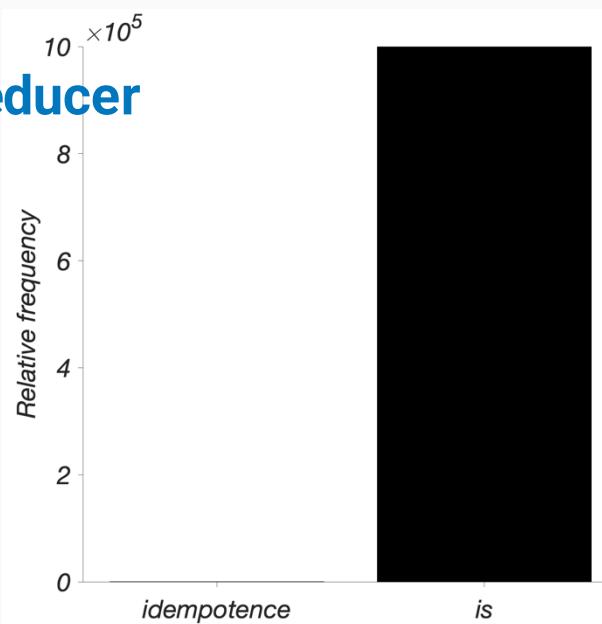
- **Don't use floating point** / real numbers for **keys**
  - Keys need to **hash** consistently, and floating point equivalence is not guaranteed
  - Use integers, strings, or tuples for keys
- Keep **map** and **reduce** simple!
  - Avoid loops if possible
  - Let sort do the work for you
- **Compare** your algorithm's complexity to the **simple / single-core solution**
  - Think about all resources: time, storage, and communication!

# Key → reducer assignment

- **All values** for a given key  $k$  go to **exactly one reducer**
- Conversely:  
A reducer acting on key  $k$  needs to see **all associated values**
- This will have consequences!

# Key-skew and idempotence

- What happens when the intermediate key distribution is **unbalanced?**
- All values for the **same key** must go to the **same reducer**
- **Different reducers** will have **different work loads**
- This is called **key skew** (or **data skew**)
  - It's a **bad thing!**
  - Bad because it is the source of much delay



# One strategy to handle key-skew: Combiners

- Key-skew leads to **high latency**
  - Reducer time scales (at least) like # values per key
- Lots of keys ⇒ lots of communication
  - **Shuffling data is expensive!**
- We can sometimes simplify the reducer's job by having mappers pre-emptively reduce (**combine**) intermediary data before shuffling.
- Necessary: Multiple mappers live on the same machine.

# Combiner example: word count

```
mapper(doc_id, doc_contents):  
    for word in doc_contents:  
        emit word, 1
```

```
combiner(word, counts):  
    partial_count = 0  
    for count in counts:  
        partial_count += count  
    emit word, partial_count
```

```
 reducer(word, counts):  
     total_count = 0  
     for count in counts:  
         total_count += count  
     emit total_count
```

Mapper node

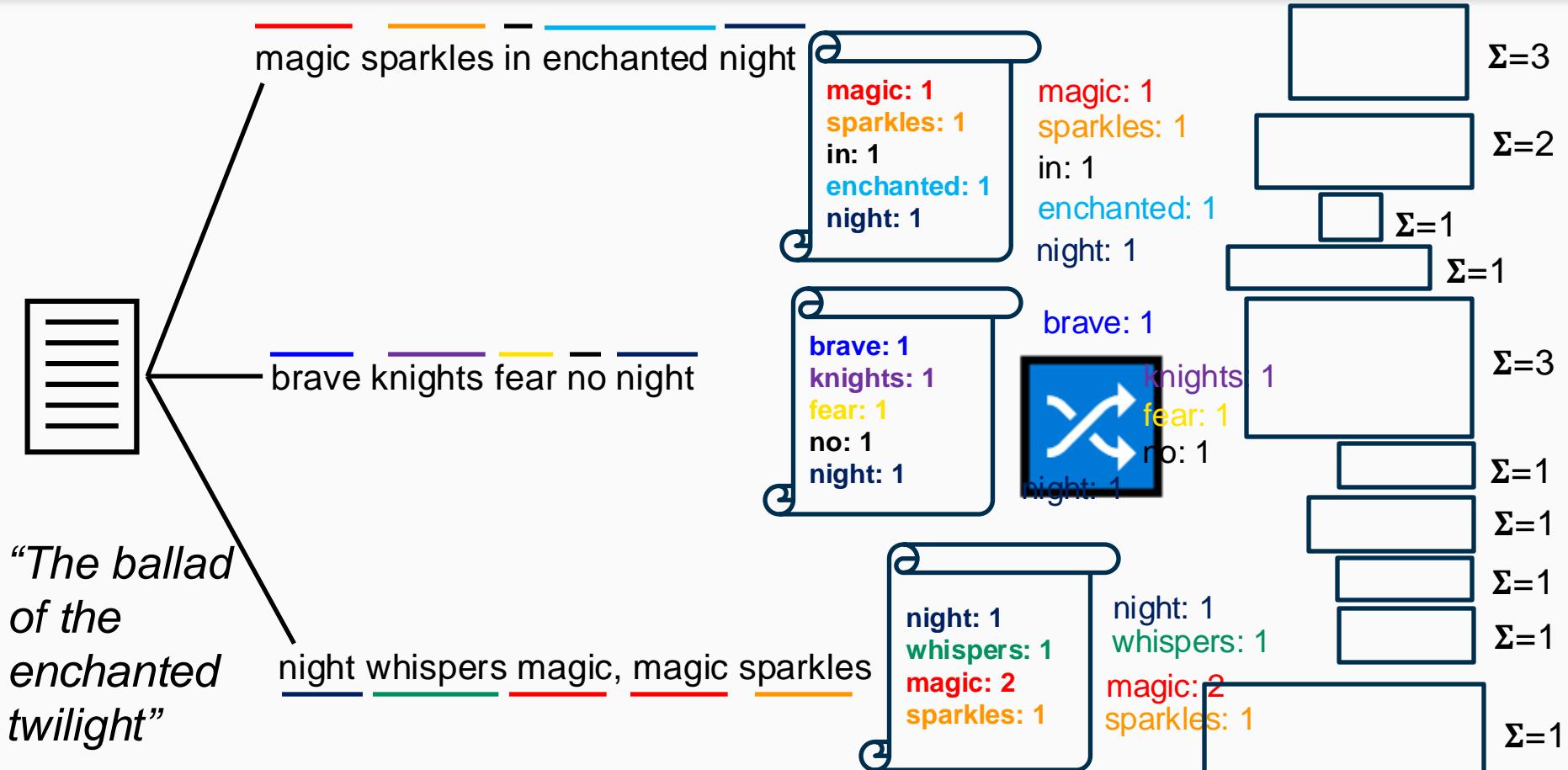
This works because summation is **commutative** and **associative**:

$$A + B = B + A$$

$$A + B + C = (A + B) + C$$

When that happens, you can re-use the **reducer** code as a **combiner**!

# What a combiner looks like in our specific example



# Heuristics for using MapReduce well

- Have **fewer mappers than inputs**
- Have **fewer reducers than intermediate keys**
- **Combiners** can help, but sometimes a fancier mapping is better
- Sometimes you can be clever with sorting to reduce communication

# Summary

- The Map-Reduce framework simplifies how we think about distributed computation (empowering through constraint), making it much more accessible
- MR was critical to the development of large-scale data analysis – particularly in the 2<sup>nd</sup> half of the 2000s.
- But it's not without drawbacks...

# Assessment of MapReduce

From a historical CS perspective

From a modern DS perspective

## The key concerns from a CS perspective (the Stonebraker criticisms)

1. Too low-level: MapReduce has no concept of a schema
2. Poor implementation: No index like RDBMS, keys = filenames
3. Not novel: previous systems used partitioning and aggregation
4. Missing important features: transactions, integrity constraints, views. It's not a database.
5. No DBMS compatibility: MapReduce ignores the rest of the ecosystem

(DeWitt & Stonebraker, 2008)

# Why was map-reduce so successful?

- DW&S raise valid points, so why was MapReduce as transformational as it was?
- Some possible reasons:
  - Simplicity: “**map**” and “**reduce**” are powerful abstractions, and often easy to write
  - Many jobs are single-shot: not worth building elaborate DB infrastructure

# Some very real problems with MapReduce

- Latency and scheduling
- Intermediate storage
- Not everything fits nicely in map-reduce
  - Iterative algorithms (e.g., gradient descent) are especially painful
  - Interactive processes (visualization, exploration) are too

# MapReduce is **\*not\*** a general purpose distributed computation engine...

- Due to its severe constraints / simplifications
- ... but it can still do a lot!
- Algorithms with **clear parallelism** and no/little looping are okay
- It works best for “one-and-done” “ballistic” tasks
- Iterative or recursive algorithms ... not so much
  - Gradient descent (and other ML approaches, e.g. alternating least squares)

# What's the role of map-reduce today?

- MapReduce is great for large  $\Omega(N)$  batch jobs that run infrequently, e.g.:
  - Data transformation / feature extraction
  - Index / data-structure construction
- It's not so great for iterative or interactive (DS) jobs:
  - Machine learning (training)
  - Search and retrieval
  - Data exploration

# So why do we study map-reduce?

- It's **historically important!** (A big step in the history of distributed computing)
- It's a **useful way of thinking** about breaking down problems into parts that can be parallelized (distributed paradigm)
- The **Hadoop ecosystem** is much bigger than map-reduce (more on this in coming weeks)
- You may **inherit legacy code.**

# Next week

- The Hadoop distributed file-system (HDFS) to manage distributed storage
- Lab 3: HDFS

# Q & R





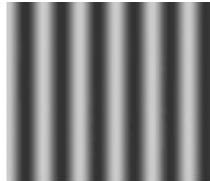
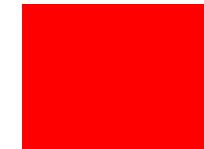
Smallest font



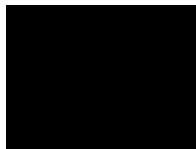
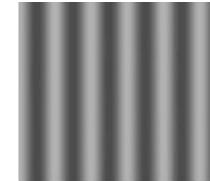
Please turn off and put  
away your cell phone



# Calibration slide



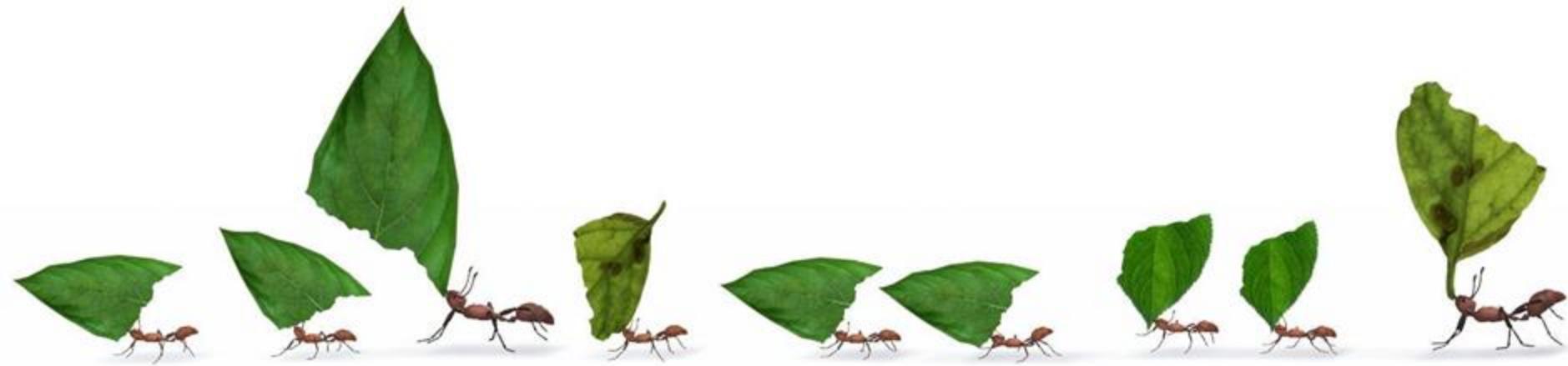
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



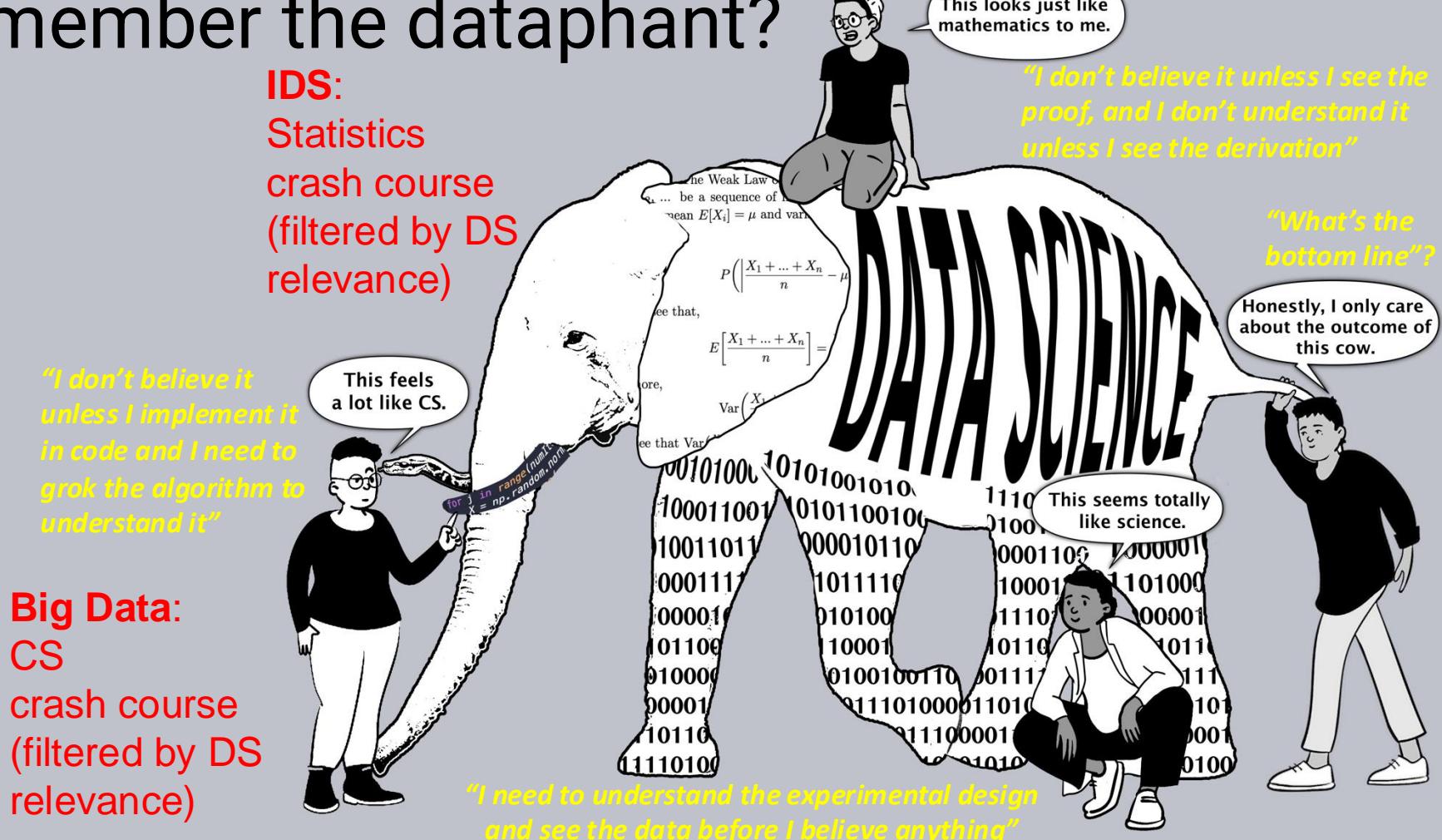
# Big Data



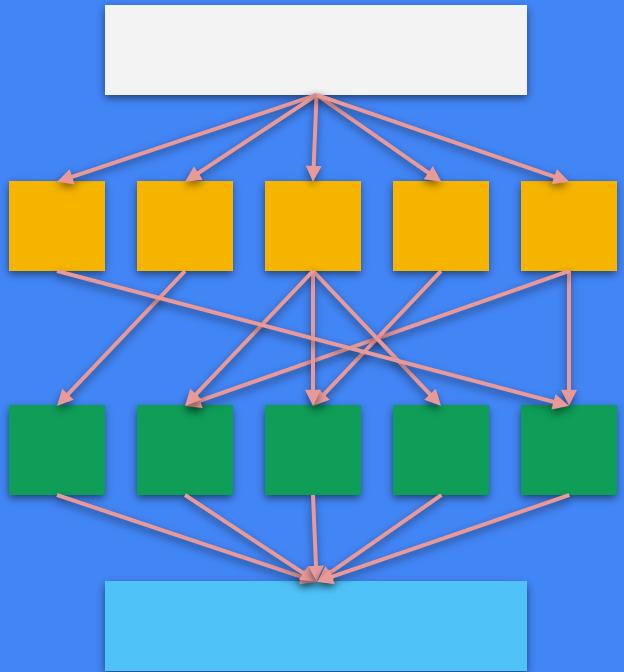
# Week 04: Distributed storage **HDFS**



# Remember the datphant?



# Last week...



# This week

1. CDS
2. Data storage
3. Distributed data storage
4. The Hadoop distributed file system (HDFS)

**\$ hadoop fs -command ...**

# Confusion, Doubt, & Struggle: Hash functions



- Functions that take an input (“the message”) and output a fixed-size string of bytes (usually a number).
- This output is the **hash value** (or “hash”)
- Hash functions have some these properties (depending on use case):
- **Deterministic** (same input → same output)
- **Fast** (to compute, for arbitrary inputs)
- Small changes in input should produce large changes in output
- **“Pre-image resistant”** (hard to invert (recover input from hash))
- **“Collision resistant”** (Different inputs should be very unlikely to produce the same output)

# Pre-image- and collision-resistance

Consider the function  $f(x) = x \bmod 10$       Is  $f(x)$  a hash function?

$$f(7) = 7 \bmod 10 = 7$$

$$f(42) = 42 \bmod 10 = 2$$

$$f(420) = 420 \bmod 10 = 0$$

$$f(2777) = 2777 \bmod 10 = 7$$

Is it pre-image resistant?

Is it collision resistant?

Consider the function  $g(x) = ([ax] + b) \bmod p$

$a, b$  = constants, e.g. 3, 5.  $p$  = prime, e.g. 97      Is  $g(x)$  a hash function?

$$g(7) = 26$$

$$g(42) = 34$$

$$g(420) = 04$$

$$g(2777) = 97$$

Is it pre-image resistant?

Is it collision resistant?

# Hashing: Why do such a thing?

There are many different and important use cases, some of which prioritize different properties of hash functions:

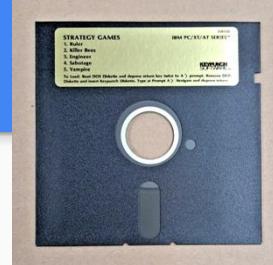
- **Error detection:** You could feed the sum of all unicode character values in a book into a hash function, e.g. one that takes the modulo of some large prime and see big differences in outcome even if only a single character was changed.
- **Cryptographic hashes:** These rely heavily on pre-image resistance. For instance, most (secure!) websites do not store your password, but only a hashed version of the password.
- **Hash tables:** These rely heavily on collision resistance. As we will see shortly, it is often more efficient to hash the (unique) keys, and storing the position in the database along with the hash, effectively creating a lookup table.
- Many others (forthcoming later in this very class)...

# 2: Data storage systems

Disks and disk arrays

# File systems and hard disks (on one computer, not distributed)

- Conceptually, file systems are made of **directories** and **files**.
- In hardware, these are stored on disks.
- Disks are made of **contiguous sectors**
  - Typically 512 to 4096 bytes
  - This is the smallest addressable unit of storage
  - Each sector belongs to at most one file
- What is the relationship between sector & file?



Floppy disk



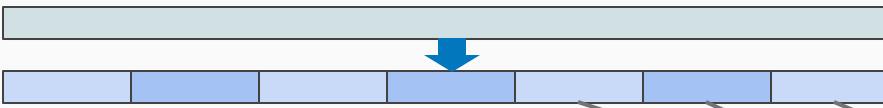
Hard disk



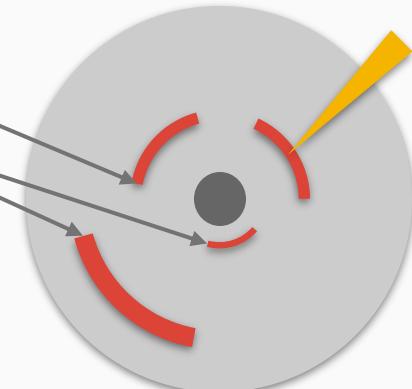
(Spinning)  
platter/disk

# Files and blocks



- **Files** are broken into **blocks**
  - Block size  $\geq$  sector size
- Each **block** is mapped onto **sectors**
- The file system (OS) hides this from us
- **Result:** a single file can spread over the entire disk

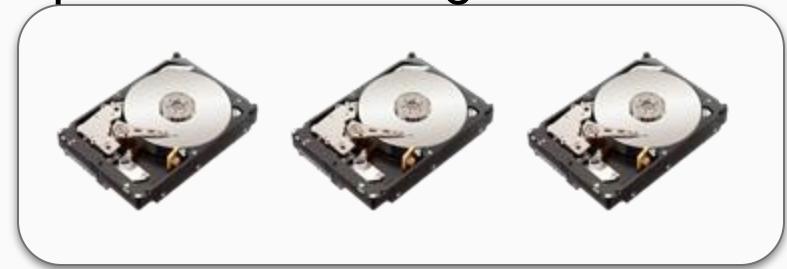
Throughput is limited by moving the **head**



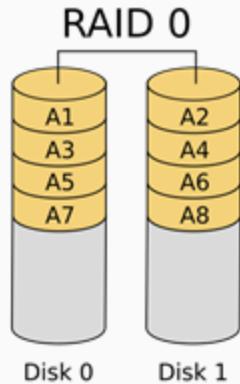
# What if our data is **too big** for a single disk?

## Redundant array of inexpensive disks (RAID)

- We could distribute files over multiple disks.
- **RAID** systems distribute storage over multiple disks in a single machine
  - But look like a single volume to the OS
- Goals of **RAID**:
  - **High Capacity**
  - **High Reliability**
  - **High Throughput**
- Comes in multiple “levels” with different reliability-capacity trade-offs
  - Note: These “levels” are not ordinal, they just make this tradeoff in different ways



# Commonly used RAID levels



Striping only

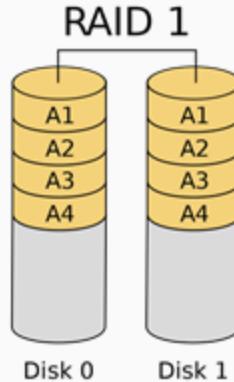
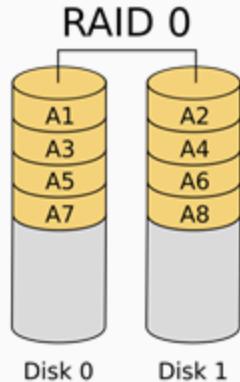
No fault-tolerance

Capacity scales linearly

# Large-scale storage: If we distribute files over multiple disks

- We need to start to worry about what happens if a **disk fails**.
- Why do we need to worry about **fault tolerance**?
- Say the chance of an individual HDD failure is low: 1 in 100 each year
- But what if we have 1000 drives?
- What is the expected value of HDD failures (assuming random and independent failure, which is the most common kind of HDD failure)?
- $EV(\text{failure/year}) = 10$

# Commonly used RAID levels



Striping only

No fault-tolerance

Capacity scales linearly

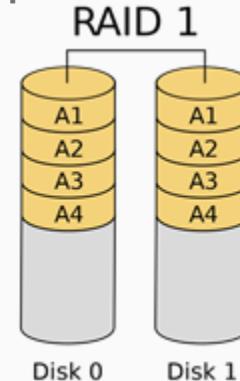
Full redundancy

(n-1) fault-tolerance

Capacity is constant

# RAID 1 assessment / tradeoff:

- Worth discussing, as some RAID lessons will be relevant later
- **Write** speed **decreases**: all data must be pushed to **all disks**
- **Read** speed can **increase**: blocks can be read in parallel from **any disk**
- *When is this trade-off worth it?*



# To understand a more commonly used tradeoff between reliability and capacity - RAID 5 – we need to make an excursion to parity bits

- A “parity bit” adds a single binary digit (bit) to a string of binary data, such that the total number of 1s is either even (“even parity”) or odd (“odd parity”).
- The primary purpose of doing so is error detection and data recovery (in the face of single bit errors – which are the most common kind).
- It does so here, by applying exclusive OR - XOR ( $\oplus$ ) :
- Say disk 0 goes down. Can we recover its data from parity?
- This logic extend beyond two disks, e.g. here for 3 disks:

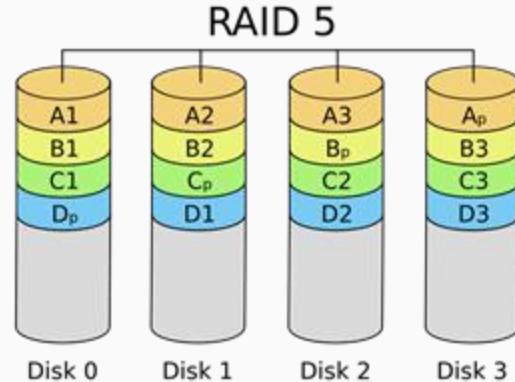
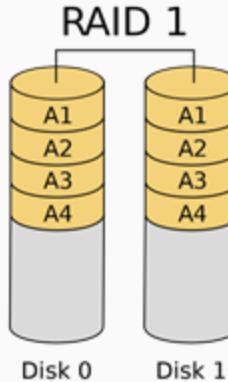
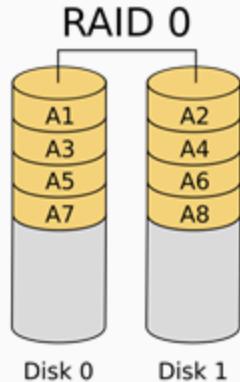
Disk 0	Disk 1	Disk 2	$0 \oplus 1 \oplus 2$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Disk 0	Disk 1	$0 \oplus 1$
0	0	0
0	1	1
1	0	1
1	1	0

What if any one disk goes down?

Can we unambiguously recover its contents from the surviving disks and the disk that contains the parity bits?

# Commonly used RAID levels



Striping only

No fault-tolerance

Capacity scales linearly

Full redundancy

(n-1) fault-tolerance

Capacity is constant

Striping with distributed **parity**

$$A_p = A_1 \text{ XOR } A_2 \text{ XOR } \dots \text{ XOR } A_{n-1}$$

Can tolerate some failure

Capacity scales almost linearly

Requires  $n \geq 3$  disks

Imagine that you are a machine learning engineer tasked with building a regression model to predict life outcomes

You want to include demographic data - like ethnicity, biological sex and immigration status - in your model.

This is reasonable, but most of demographic data is inherently qualitative in nature:

Immigration status	Code
Citizen	1
Resident	2
Other	3

Challenging, because the raw predictor data has none of the properties multiple regression needs

Python will do it, but results will be nonsensical

Can we do it?

How?

# One-hot encoding! (“Dummy coding”, properly done)

Immigration status	Code	OHE	Immigration status	Citizen?	Resident?	Other?
Citizen	1		No	0	0	0
Resident	2		Yes	1	1	1
Other	3					

Should we include all k (here 3) columns in the model, so as not to lose information?

 Careful! We fell into the “**dummy variable trap**” 

Why is that catastrophically bad?

It \*guarantees\* that the model has an extremely severe multi-collinearity problem

The solution?

One-hot encoding with **\*any\*** k-1 columns:

Immigration status	Code	OHE	Immigration status	Citizen?	Resident?
Citizen	1		No	0	0
Resident	2		Yes	1	1
Other	3				

# Why is multi-collinearity such a problem for regression models?

A system of linear equations:

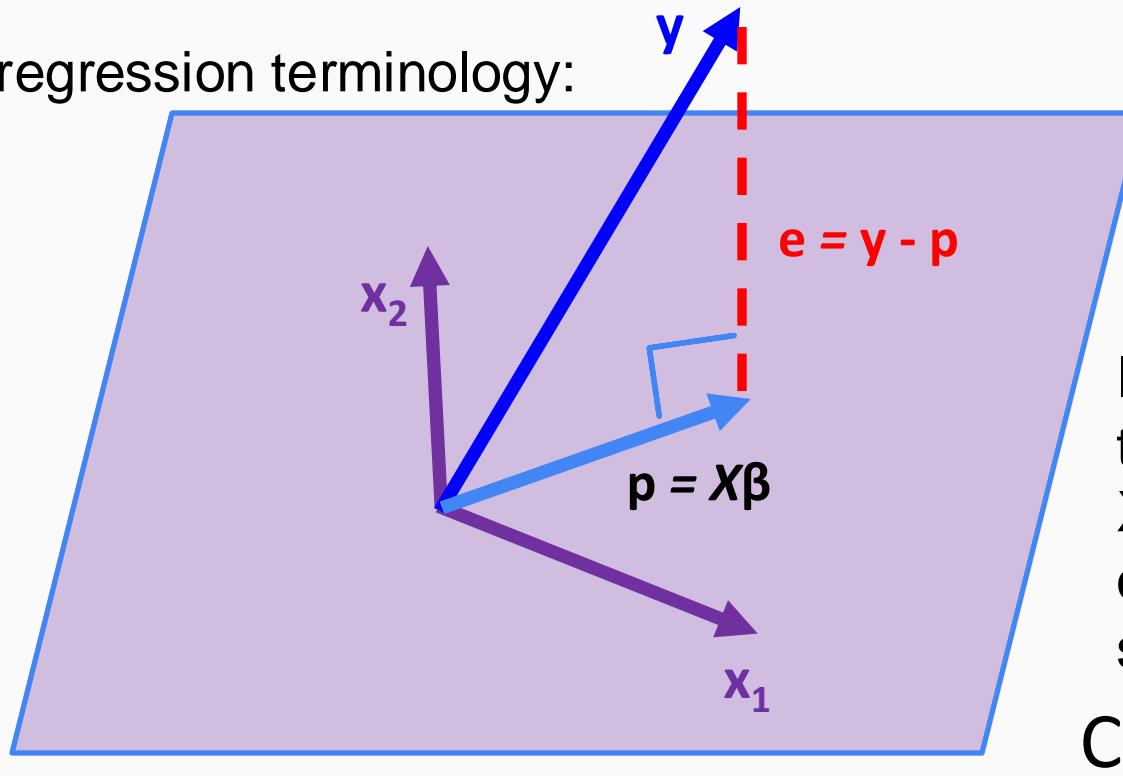
$$Ax = b$$

$$X = [x_1 \ x_2]$$

In linear regression terminology:

$$X\beta = y$$

$$X\beta - y = 0$$



Insight:  $y$  is not in the column space of  $X$ . But the projection of  $y$  onto the column space is.

$$C(X)$$

If  $x_1$  and  $x_2$  are pointing in similar directions, projecting on a plane becomes “wobbly” - the  $\beta$  estimates become unreliable:

A system of linear equations:

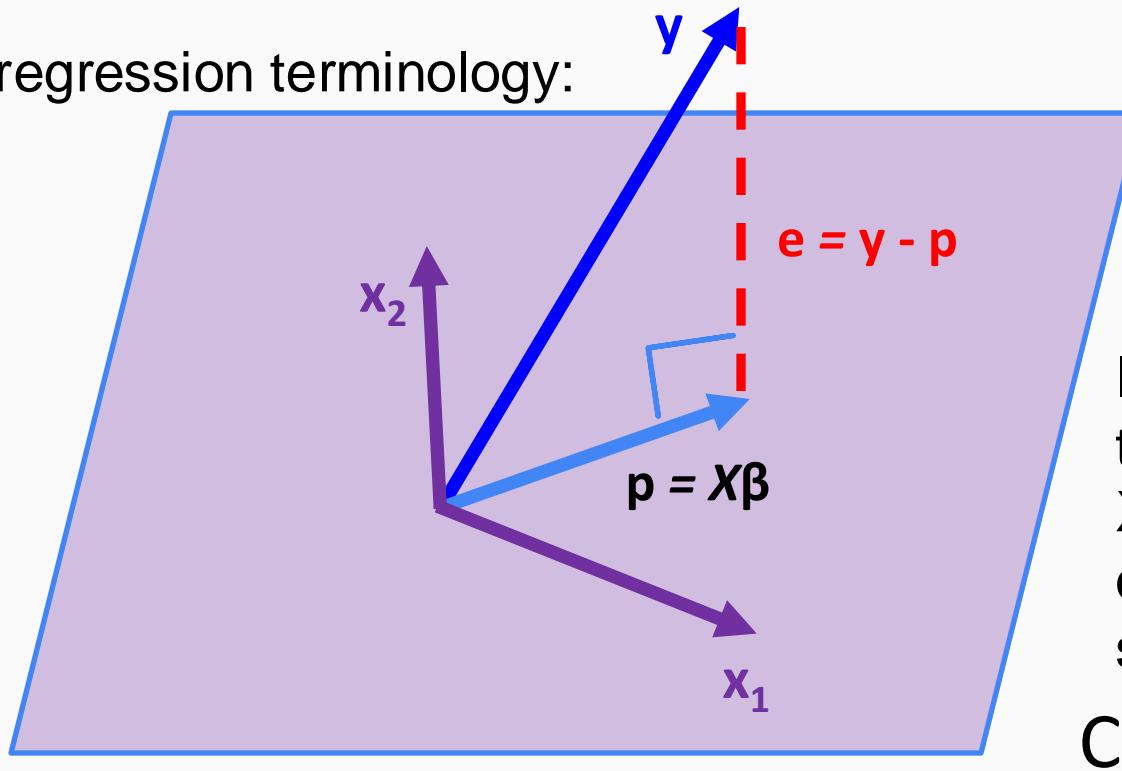
$$Ax = b$$

$$X = [x_1 \ x_2]$$

In linear regression terminology:

$$X\beta = y$$

$$X\beta - y = 0$$

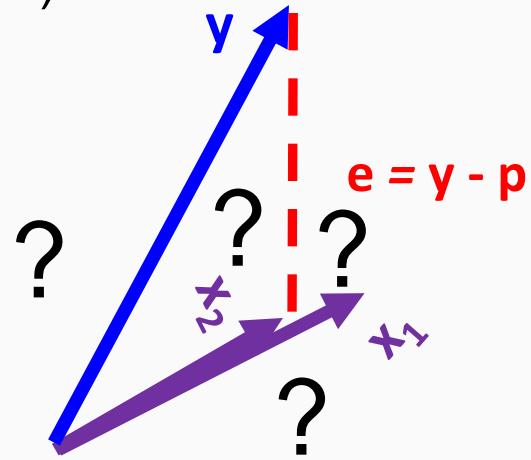


Insight:  $y$  is not in the column space of  $X$ . But the projection of  $y$  onto the column space is.

$$C(X)$$

If  $x_1$  and  $x_2$  are pointing in similar directions, projecting on a plane becomes “wobbly” - the  $\beta$  estimates become unreliable:

Estimating the  $\beta$ s will become very unreliable, as the plane spanned by these basis vectors becomes very sensitive to minute perturbations (from noise, specific train/test splits, etc.)



We effectively inject excess variance (in terms of the bias/variance tradeoff) in the model, for no obvious gain and no good reason. The model became objectively worse.

# The value of a DS approach: Integrating multiple perspectives

CS perspective:  
Parity bits  
A useful feature

Stats perspective:  
Dummy variable trap  
A terrible bug

DS perspective:  
**Redundancy overcompleteness principle**  
Bug or feature depending on use case

# Back to large file storage: Features of RAID

- **Striping blocks** over multiple drives increases **capacity and throughput**
  - **Not** reliability
- **Striping blocks** ⇒ a file can be larger than any single drive
- Adding **parity blocks** improves reliability
  - If a data block is corrupted or lost, it can be recovered from other blocks + parity
  - If a parity block is corrupted or lost, it can be recomputed from data blocks

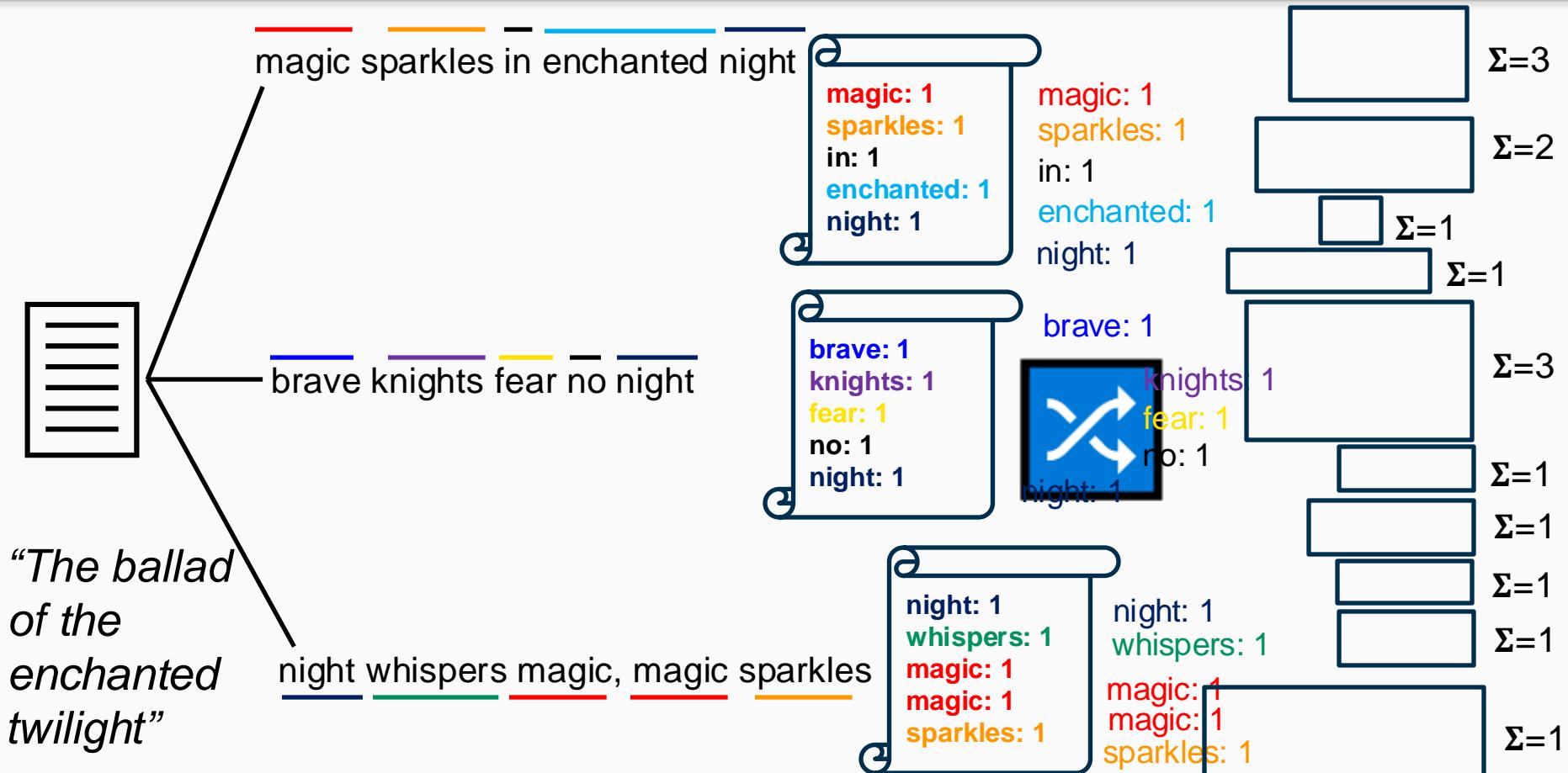
# 3: Distributed data storage

Over many computers

# Why wasn't RAID enough?

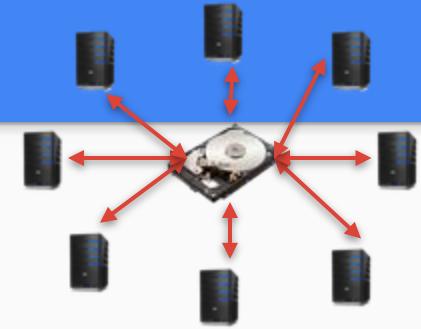
- RAID improves capacity, fault-tolerance, and (read) throughput **on a single machine**
- What about distributed \*computation\*?
  - **Communication** over the network is a **bottleneck**
- *What are the common access patterns?*
  - Can we do **better** than both **fully localized** and **fully distributed**?

# Reminder: How mapReduce works for word counts



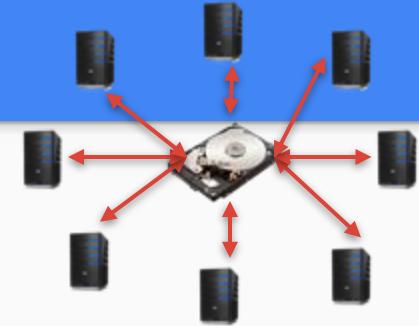
# Let's start simple

- Imagine implementing Map-Reduce from scratch
  - ... with all data located on a central file server (“center node”)
- Center node sends (**mapper**, **reducer**) code to each processor **+ block of data**
- Processors send **output** back to center node
  - **Mappers** → intermediate results
  - **Reducers** → final results



# Let's start simple

- Imagine implementing Map-Reduce from scratch
  - ... with all data located on a central file server ("center node")
- Center node sends (**mapper**, **reducer**) code to each processor **+ block of data**
- Processors send **output** back to center
  - **Mappers** → intermediate results
  - **Reducers** → final results



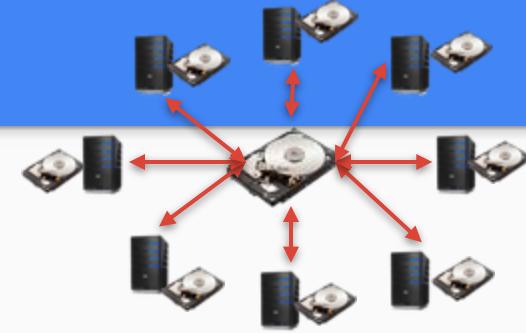
This will work, but it's inefficient (slow)!

Each job moves the entire data set over the network!

This is a **failure of data locality**.

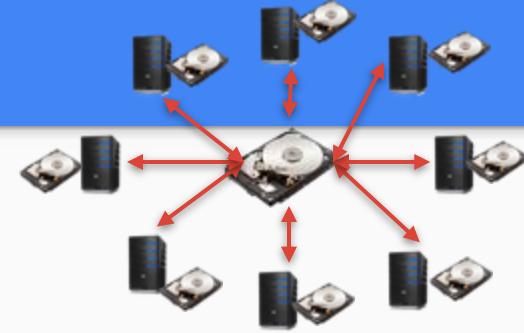
# A better way: Localize (distribute) all data?

- What if all data is replicated on all processor nodes?
- Central node sends (**mapper**, **reducer**) code to each processor  
**+ id of data block**
- Processor send **output** back to Center node
  - **Mappers** → intermediate results
  - **Reducers** → final results



# A better way: Localize (distribute) all data?

- What if all data is replicated on all processor nodes?
- Central node sends (**mapper**, **reducer**) code to each processor  
**+ id of data block**
- Processor send **output** back to Center node
  - **Mappers** → intermediate results
  - **Reducers** → final results



This also will work, but it's expensive, due to extreme redundancy!

Each worker needs a large amount of storage.

Most workers don't touch most of the data (inefficient storage)

# Scalability principles of distributed work



# Distributed file systems

- **Distributed file systems** store data over many machines
- They strike a more reasonable tradeoff between the need to minimize communication over the network and redundancy due to localized storage.
- There are other design considerations...

# Design considerations

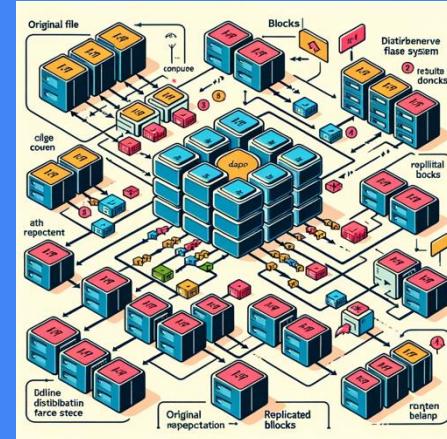
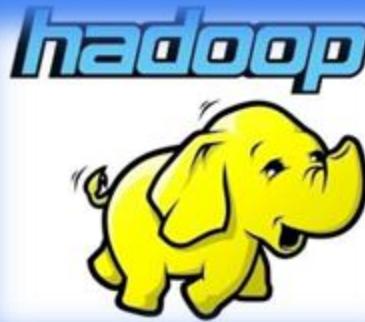
- Communication costs (bytes transferred)
- Fault tolerance
- Redundancy vs. communication
- Granularity of access (block size)
- Data Locality
- Common access patterns
- **Programs are small, data is big!**

# 4: The Hadoop distributed file system **(HDFS)**

# Hadoop distributed file system (HDFS)

- HDFS is the storage component of Hadoop
  - **Useful beyond map-reduce!**
- Provides distributed, redundant storage
- Optimizes for **single-write, multiple-read** patterns
  - Typical of map-reduce applications

# The Hadoop framework



**MapReduce**

*Processing engine*

**YARN**

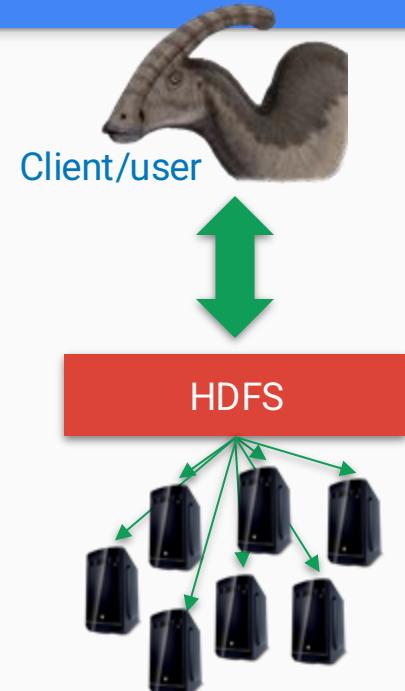
*Resource manager*

**HDFS**

*Distributed file system*

# Using HDFS

- HDFS is a “file system”, but we use it differently than the one you are used to from your computer.
- HDFS sits on top of the operating system’s built-in file system
- Better to think of it as an **application** that stores files for you
  - Kind of like Google Drive or Dropbox (that provide controlled data access)
  - Data can be accessed through the “**hadoop fs**” command



# Two types of nodes (= computers) in HDFS



Name node



Data nodes

# The name node

- Clients talk to the name node to **locate data**
  - Analogous to the file system (but not storage device) in a standard OS
- Name node knows the mappings of:
  - **Files** → **blocks**
  - **Blocks** → **data nodes**
- Keeps a record of transactions
  - Backed up remotely for durability



Client/user

# “Data” nodes (also where mappers and reducers are run)

- Stores each block as two files in the local file system:
  - **Data block** (variable size up to defined max, typically 128MB)
  - Metadata:
    - **Checksum**
    - **Generation stamp**
- **Checksum:** From hashing, used to detect storage errors
  - Analogous to **parity blocks** in RAID, but replicated on all nodes
  - Weaker than parity: can detect errors, but not correct them
- **Generation stamp:** used to detect updates



# Division of labor/responsibilities

- Name nodes do not store **data!**
- Data nodes do not store file names/file structure!
- Name node failure is **catastrophic**
- Data node failure can be tolerated, up to a point
  - Depends on how many replications of each file you have

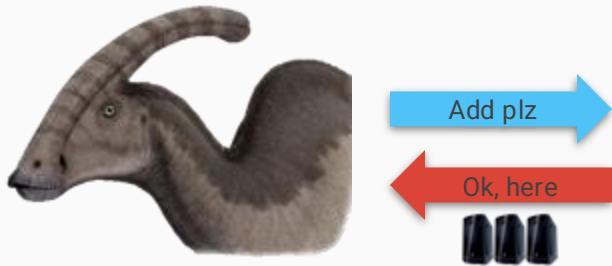


## Example operation: writing to HDFS



1. Client wants to [add a block](#)

## Example operation: writing to HDFS



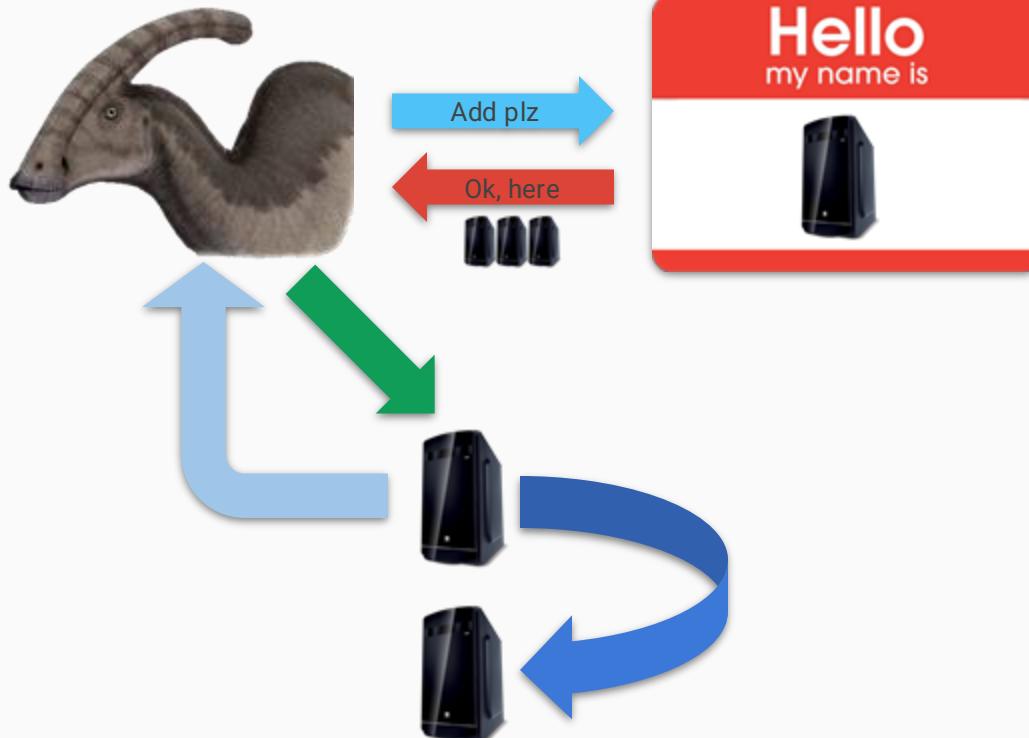
1. Client wants to **add a block**
  - a. Name node **responds** with a list of data nodes

## Example operation: writing to HDFS



1. Client wants to **add a block**
  - a. Name node **responds** with a list of data nodes
2. Client **sends block** to DN1

## Example operation: writing to HDFS



1. Client wants to **add a block**
  - a. Name node **responds** with a list of data nodes
2. Client **sends block** to DN1
3. DN1 stores, **acknowledges**, and **sends block** to DN2

## Example operation: writing to HDFS



1. Client wants to **add a block**
  - a. Name node **responds** with a list of data nodes
2. Client **sends block** to DN1
3. DN1 stores, **acknowledges**, and **sends block** to DN2
4. DN2 stores, **acknowledges**, and **sends block** to DN3

## Example operation: writing to HDFS

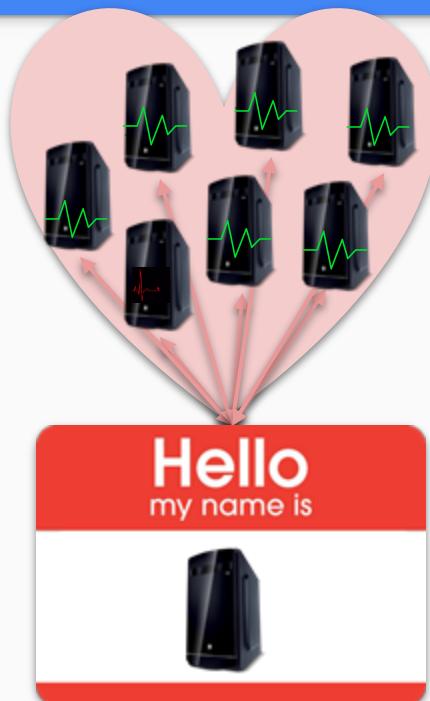


1. Client wants to **add a block**
  - a. Name node **responds** with a list of data nodes
2. Client **sends block** to DN1
3. DN1 stores, **acknowledges**, and **sends block** to DN2
4. DN2 stores, **acknowledges**, and **sends block** to DN3
5. DN3 stores and **acknowledges**

Add complete, close file, **alert name node**

# Name and data node communication

- Data nodes periodically signal the name node
  - “Heartbeat” 
- The name node always knows which data nodes are alive
  - At least, within the last 3 seconds
  - Name node can infer failures and insufficient replication
- Name node may respond with update messages
  - E.g.: replicate block x from data node y



# Recovering from name node failure: Checkpoints

- Checkpoints are snapshots of the current name node's state
  - Directory structure, block maps, and journal
  - Name node keeps all of this information in RAM
- These are created periodically to ensure fast recovery when NN fails
- Checkpoints cannot be updated, only replaced

# HDFS is \*not\* "POSIX"-compliant

- POSIX: "Portable Operating System Interface" maintained by IEEE
- Important non-compliance: Updates are append-only
  - No changing old data!
  - What written is done.
  - This makes replication logic much simpler
- Not all file modes are supported
  - Not all modes make sense in this limited context anyway
  - E.g., executable (neither desirable nor necessary for mapReduce)



# Why doesn't HDFS work like the file system on my desktop?

- Desktop computing needs to support all kinds of uses
  - E.g. thousands of small configuration files
  - Files that update frequently (e.g., browser cache)
- Large data analysis jobs have different needs
  - Few, large files
  - Frequent read access (analysis)
  - Infrequent updates (append-only can be okay)

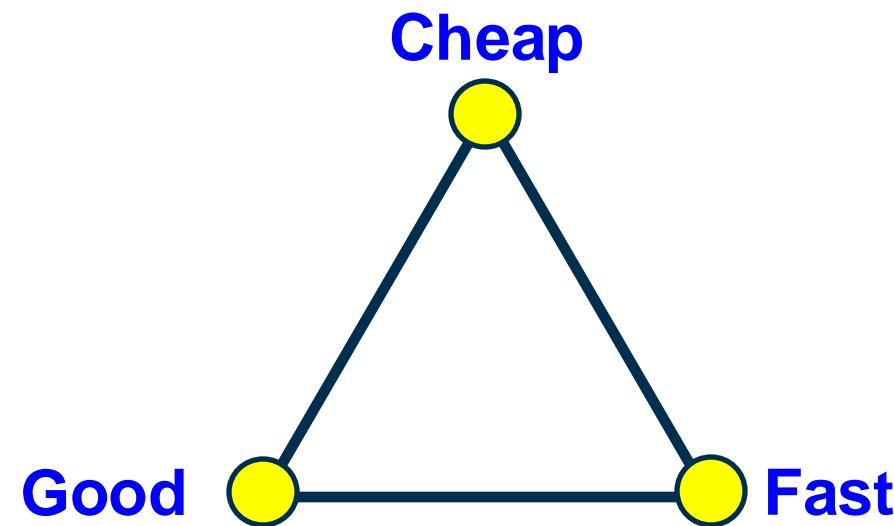
# HDFS

and

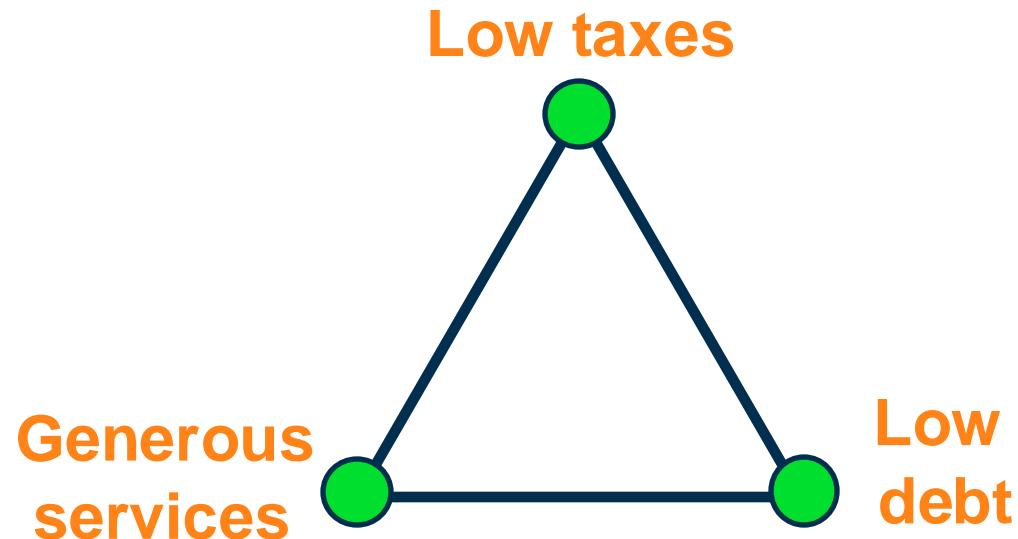
# CAP

# Popular trilemmas\*

Project management:



Governance:

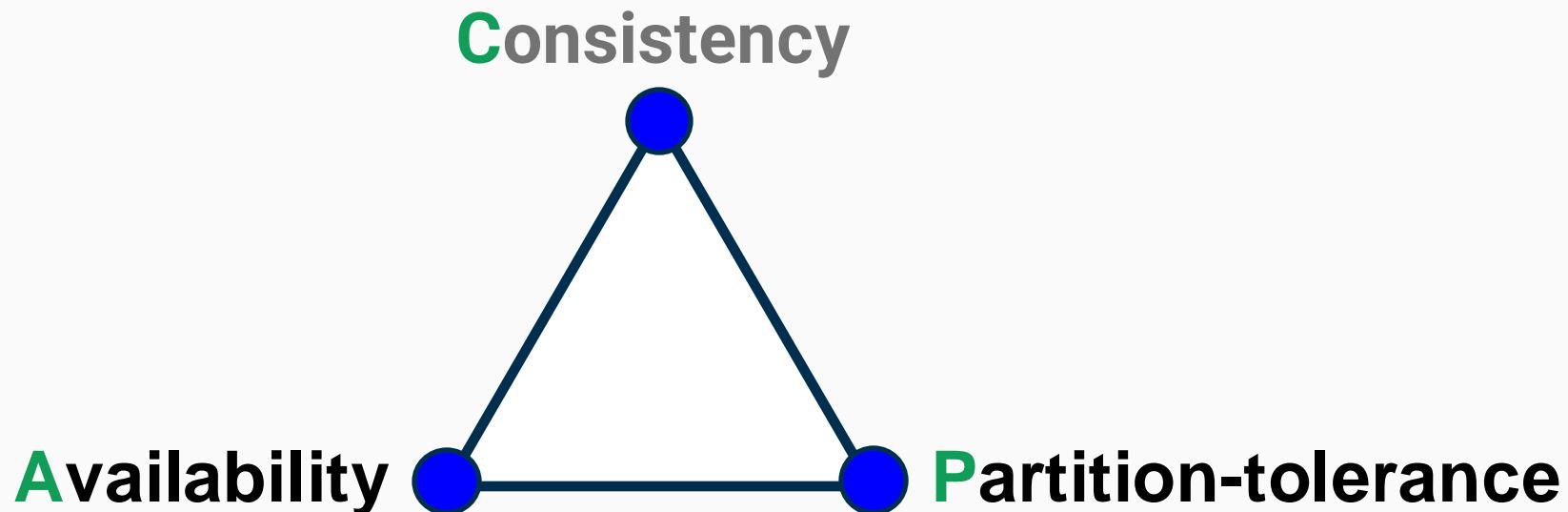


You can pick the nodes along ONE edge

\*Assuming no (zero) corruption

# The CAP theorem for distributed storage

[Brewer 1998, Gilbert & Lynch 2002]



# The CAP theorem for distributed storage in practice

[Brewer 1998, Gilbert & Lynch 2002]

- **Consistency:**

- Read always produces the most recent value

- **Availability:**

- Requests cannot be ignored

- **Partition-tolerance:**

- System maintains correctness during network failure



Pick which one  
you prefer

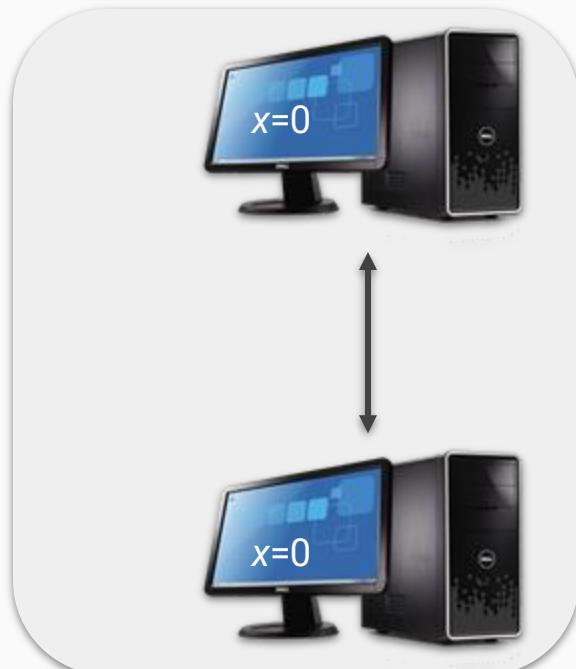
← Gotta have this one

# Why can't we have CAP?

## Assume that we could...



VelociCAPtor



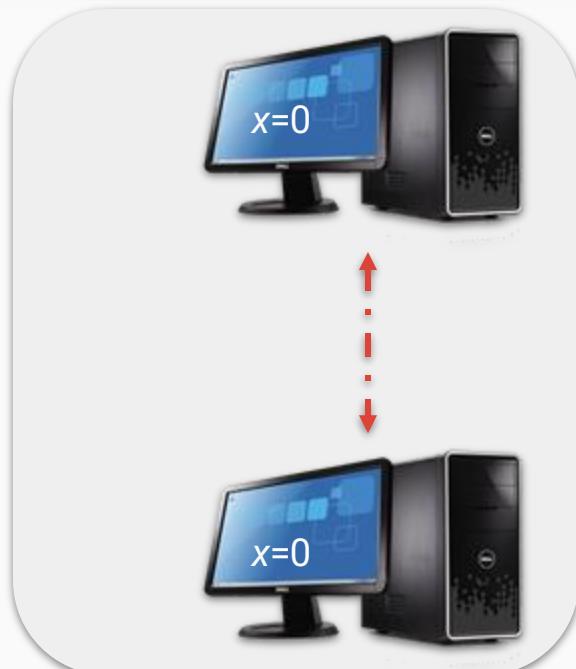
1.  $M_1$  and  $M_2$  initialized with  $x=0$

# Why can't we have CAP?

## Assume that we could...



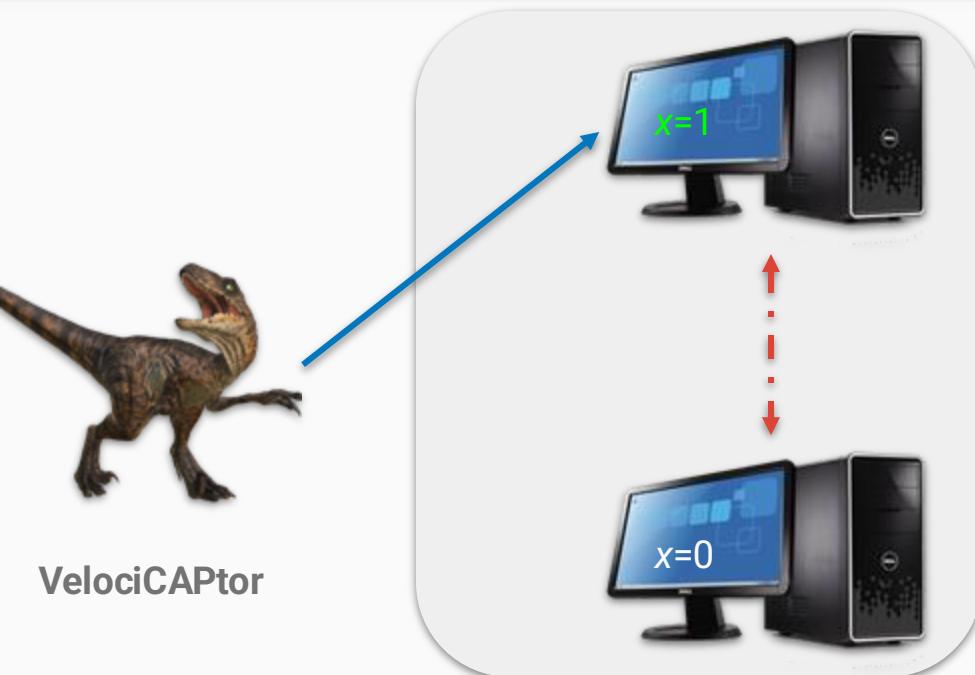
VelociCAPtor



1.  $M_1$  and  $M_2$  initialized with  $x=0$
2. Network fails

# Why can't we have CAP?

## Assume that we could...

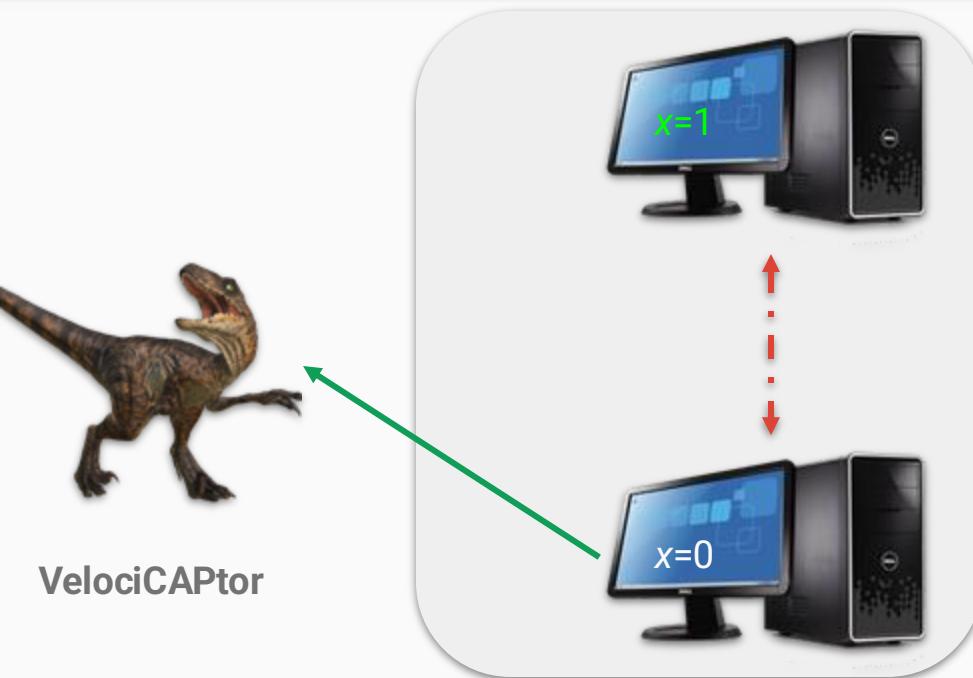


VelociCAPtor

1.  $M_1$  and  $M_2$  initialized with  $x=0$
2. Network fails
3. Update  $x=1$  on  $M_1$

# Why can't we have CAP?

## Assume that we could...



1.  $M_1$  and  $M_2$  initialized with  $x=0$
2. Network fails
3. Update  $x=1$  on  $M_1$
4. Read  $x$  from  $M_2$

What happens?

# How does HDFS handle CAP?

- **Consistency**
  - Centralized name node always has a consistent view of the file system
  - Data can be added (appended), but **not modified!**
- **Availability**
  - Yes, as long as the name node works
  - If the name node goes offline, we're out of luck
- **Partition-tolerance**
  - Yes, but it depends on the network configuration and replication factor

# Wrap-up on HDFS

- Files divide into blocks, and are replicated across the cluster
- Name node allocates blocks and interfaces with clients
- Blocks are append-only ⇒ optimized for write-once, read-many patterns

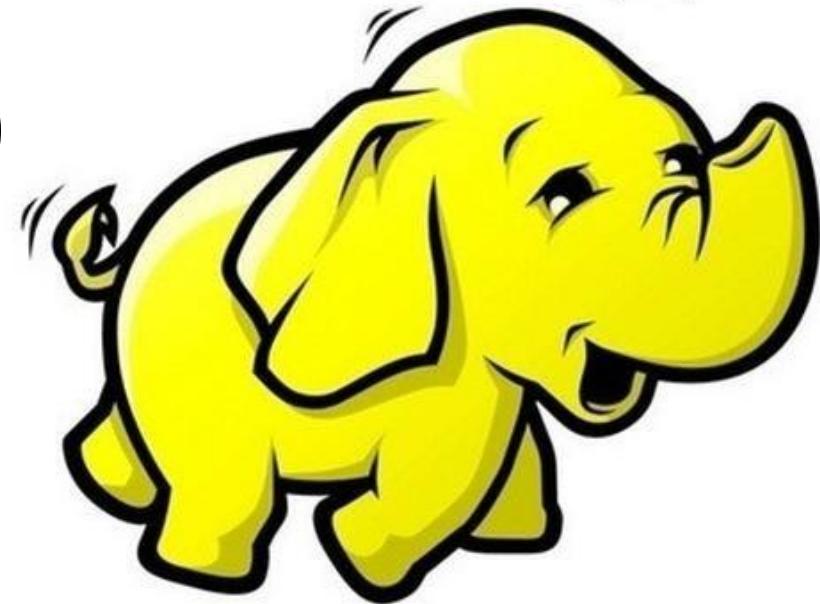
# Next week

We will cover big data infrastructure (both in general and at NYU specifically)

**Think of questions!**

*hadoop*

Q & R





# NYU HPC Big Data

[hpc@nyu.edu](mailto:hpc@nyu.edu)

<https://hpc.nyu.edu>

# Who We are?

Research Technology Services Team, IT

- High Performance Computing (HPC) for research and courses
- Support Big Data, ML/DL/AI
- Cloud Computing with Google Cloud Platform (GCP), Amazon Web Service (AWS)
- JupyterHub for courses
- Security Data Research Environment (SDRE)
- High Speed Research Network (HSRN)
- Other research related requests for IT supports

# Outline

- Dataproc cluster for courses
  - Available services with Dataproc
  - Backend setups for Dataproc
- Big data on HPC Greene cluster
  - Spark with single node and multiple nodes cluster mode
  - Dask on Greene cluster
- Q&A

# What is Dataproc

- Dataproc is a managed Hadoop and Spark service in Google Cloud Platform. It takes advantage of open source tools for big data processing and machine learning.
- Its components include Hadoop Distributed File System (HDFS), MapReduce, Spark, Hive, Trino, Pig etc.
- The default block size in HDFS is 128 MB. The file system is not good for storing many small sized files.
- YARN is used for resource management, and job scheduling and monitoring. Its functionalities are very preliminary, comparing to its HPC analog such as SLURM.

# Dataproc For Courses I

- The cluster serves many hundreds of course takers, for practice and doing exercise, not for real big data projects. The cluster is created at the beginning of a semester and destroyed at the end of a semester
- Spark runs on top of YARN
- Spark structured streaming is feasible

# Dataproc For Courses II

- User entrance
  - <https://dataproc.hpc.nyu.edu/>
- Getting started user guide
- Home filesystem
  - When you first login, you land at /home/<netid>\_nyu\_edu directory
  - Where you can run the commands: ls, pwd, cd etc
- HDFS filesystem
  - 500GB quota per user
  - Access by run the commands: hdfs dfs -ls, hdfs dfs -put ...

# Demos

## NYU Dataproc

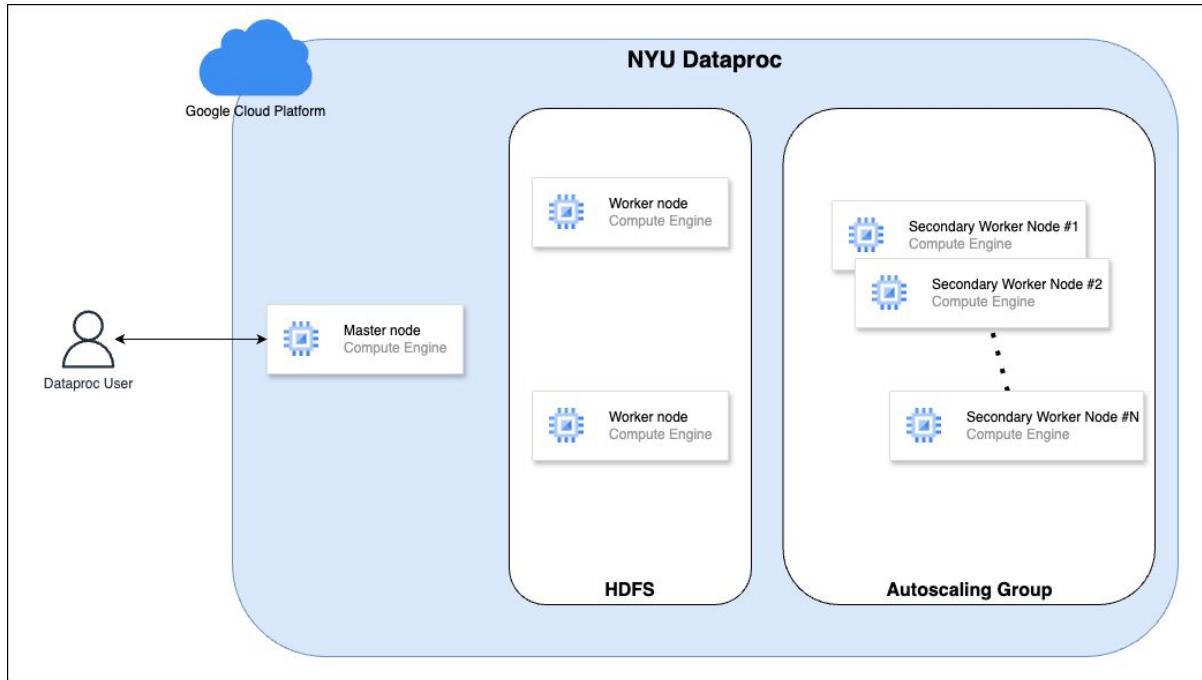
[Google Cloud Dataproc](#) is a cloud-based Hadoop service that NYU HPC provides for courses. It comes with the following interfaces for interacting with Hadoop:

Interface	Description	URL
<b>Command Line Interface Console</b>	This is how you log into Dataproc.	<a href="https://dataproc.hpc.nyu.edu/ssh">https://dataproc.hpc.nyu.edu/ssh</a>
<b>Data Ingest Console</b>	This is how you upload data into Dataproc.	<a href="https://dataproc.hpc.nyu.edu/ingest">https://dataproc.hpc.nyu.edu/ingest</a>
<b>MapReduce Job History</b>	A web interface where you can see all MapReduce jobs that have run on the cluster.	<a href="https://dataproc.hpc.nyu.edu/jobhistory/">https://dataproc.hpc.nyu.edu/jobhistory/</a>
<b>Spark History Server</b>	A web interface where you can see all Spark jobs that have run on the cluster.	<a href="https://dataproc.hpc.nyu.edu/sparkhistory/">https://dataproc.hpc.nyu.edu/sparkhistory/</a>
<b>YARN Application Timeline</b>	A web interface where you can see all applications (both Hadoop MapReduce and Spark) that have run on the cluster.	<a href="https://dataproc.hpc.nyu.edu/apphistory/">https://dataproc.hpc.nyu.edu/apphistory/</a>
<b>YARN ResourceManager</b>	YARN is the resource manager and job scheduler used by the Dataproc cluster. YARN allows you to use various data processing engines for batch, interactive, and real-time stream processing of data stored in HDFS. The YARN web interface allows you to see all current and recently executed applications, as well as information about the current state of the cluster (such as number of vCores and RAM free).	<a href="https://dataproc.hpc.nyu.edu/yarn/">https://dataproc.hpc.nyu.edu/yarn/</a>

# Dataproc For Courses III

- Maximum YARN application lifetime is **5** hours. Vast majority finish in <5 mins.
- The default Spark deploy mode is cluster, that is recommended for production run. For debugging purposes, run spark in the client mode:
  - spark-shell --deploy-mode client
- login node is for debugging and submitting jobs, not for computing heavy work. When done for the day, please log out.
  - Memory limit per user is **3** GB
  - User process lifetime is **48** hours.
- pyspark is available, and the python and its package are from conda install.
- container log size check enabled. App outputs should be written to HDFS, not to stdout.

# Dataproc System Setup I



# Dataproc System Setup II

- Login to master node with your NYU Google account
  - home folder name: **netid\_nyu\_edu**
- Start with 1TB Hadoop Distributed File System (HDFS). Scale up to **8TB**.
- Ingress storage bucket to upload large input data
- All files in home and HDFS will be **deleted** at the end of the semester

# **Big Data On Greene HPC Cluster**

# Greene Cluster

- Greene is a general purpose HPC cluster for NYU public use
- Greene hardware
  - ~38K CPU cores, 220TB memory
  - 768 GPUs, 13TB GPU memory
  - 12PB parallel storages
  - HDR 200Gb/s Infiniband network for computation communications and storages

<https://hpc.nyu.edu>

# Greene Cluster in Research Computing Center (RCDC)



# Greene Cluster in Research Computing Center (RCDC)

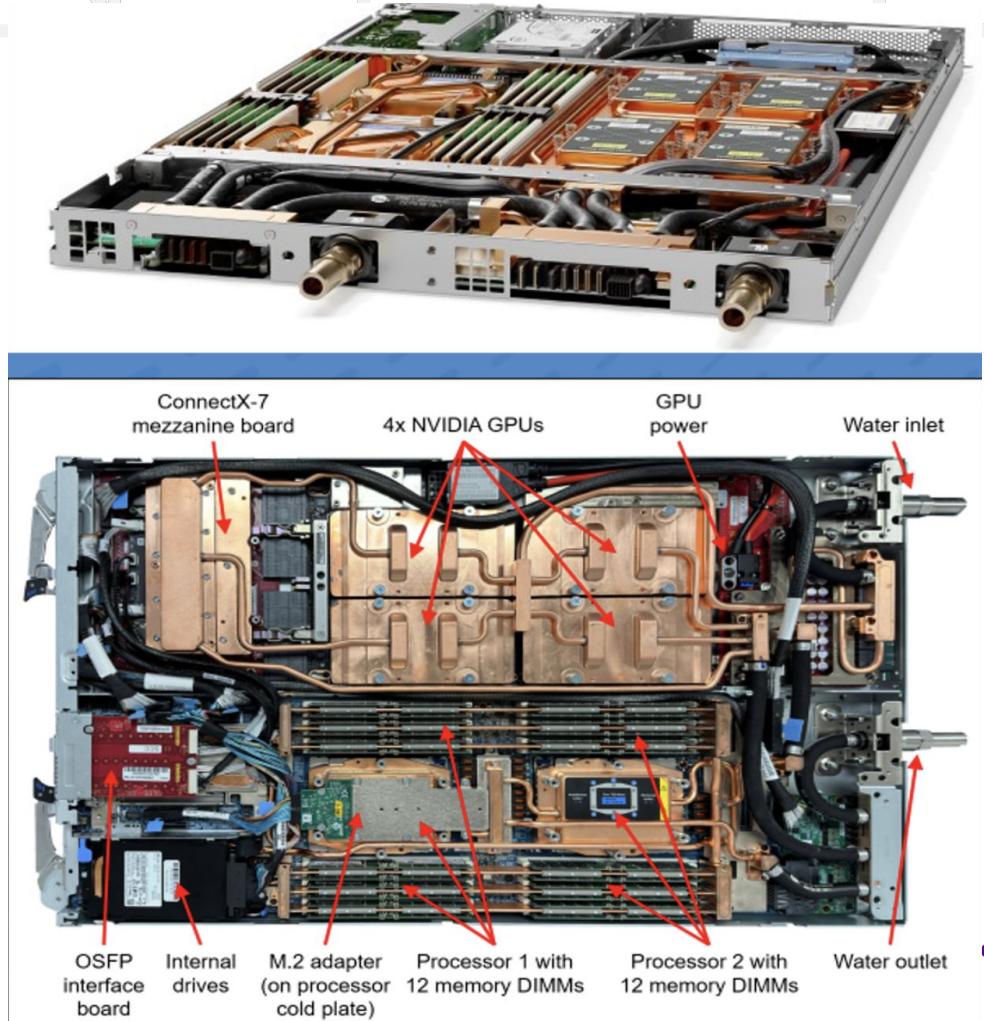


# Greene Cluster in Research Computing Center (RCDC)



# Greene Cluster in Research Computing Center (RCDC)

Liquid cooling NVIDIA H100 GPU server SD650N-V3



# Before Logging In

## Check your Network Location

### **NYU Campus:**

From within the NYU network, that is, from an on-campus location, or after you VPN inside NYU's network, you can follow the next page for instructions to login to the HPC clusters directly.

### **Off Campus:**

From a commercial network outside of NYU (from an off campus location). Before following the next page to log into the clusters, you would need to connect to the NYU VPN or the HPC Gateway Server.

# Greene Cluster Logins

- Hostname: greene.hpc.nyu.edu
- 3 login nodes are configured with load balancer, open to NYU network and NYU VPN
- HPC clusters are using NYU netID and associated password
- Mac/Linux/Windows Subsystems from terminal

```
ssh NYUnetID@greene.hpc.nyu.edu
```

- For Windows: Putty, MobaXterm
- VScode

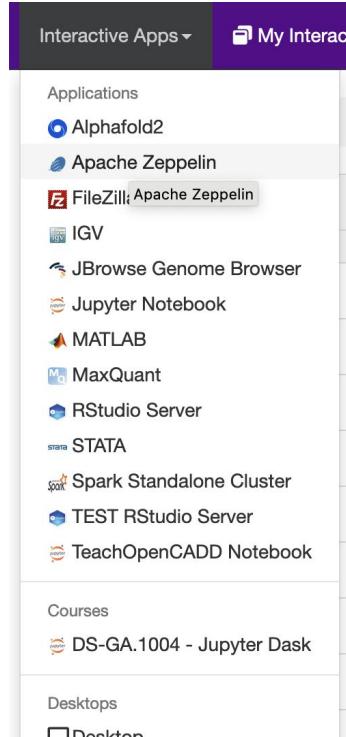
# Greene Open OnDemand Server

- HPC cluster in browser
  - <https://ood.hpc.nyu.edu>
- Provide GUI interface to Jupyter notebooks, Matlab, **Spark Standalone Cluster, Dask with Jupyter, Remote Desktop**

The screenshot shows a user interface for managing interactive applications. At the top, there are tabs for "Interactive Apps" and "My Interactive". Below this is a sidebar with sections for "Applications", "Courses", and "Desktops". The "Applications" section lists several services with their icons: AlphaFold2 (blue circle), Apache Zeppelin (blue square), FileZilla (red square with a white F), IGV (blue square with a white bar chart), JBrowse Genome Browser (blue square with a white genome icon), Jupyter Notebook (orange square with a white notebook icon), MATLAB (blue square with a white MATLAB logo), MaxQuant (blue square with a white mass spec icon), RStudio Server (blue square with a white RStudio logo), and STATA (blue square with a white STATA logo). The "FileZilla; Apache Zeppelin" entry is highlighted with a gray background. The "Courses" section contains one item: DS-GA.1004 - Jupyter Dask. The "Desktops" section contains two items: Desktop and Desktop + Metashape.

Section	Item
Applications	AlphaFold2
	Apache Zeppelin
	FileZilla; Apache Zeppelin
	IGV
	JBrowse Genome Browser
	Jupyter Notebook
	MATLAB
	MaxQuant
	RStudio Server
	STATA
Spark Standalone Cluster	
TEST RStudio Server	
TeachOpenCADD Notebook	
Courses	DS-GA.1004 - Jupyter Dask
Desktops	Desktop
	Desktop + Metashape

# Spark on OOD Demons



The screenshot shows a user interface for managing interactive applications. At the top, there's a dark header bar with the text "Interactive Apps" and "My Interactive". Below this is a list of applications, each with a small icon and a name. The applications listed are:

- AlphaFold2
- Apache Zeppelin
- FileZilla; Apache Zeppelin
- IGV
- JBrowse Genome Browser
- Jupyter Notebook
- MATLAB
- MaxQuant
- RStudio Server
- STATA
- Spark Standalone Cluster
- TEST RStudio Server
- TeachOpenCADD Notebook

Below the application list, there are sections for "Courses" and "Desktops", each containing a single item.

# Dask on OOD Demos

The screenshot shows a user interface for managing interactive applications. At the top, there are two tabs: "Interactive Apps" (selected) and "My Interactive Apps". Below the tabs is a list of applications categorized into sections: Applications, Courses, and Desktops.

- Applications:**
  - AlphaFold2
  - Apache Zeppelin
  - FileZilla
  - IGV
  - JBrowse Genome Browser
  - Jupyter Notebook
  - MATLAB
  - MaxQuant
  - RStudio Server
  - STATA
  - Spark Standalone Cluster
  - TEST RStudio Server
  - TeachOpenCADD Notebook
- Courses:**
  - DS-GA.1004 - Jupyter Dask
- Desktops:**

# Spark Jobs in Batch Mode

Examples are available:

/scratch/work/public/apps/pyspark/3.5.0/examples/spark

# Greene Compute Nodes

Nodes	Number	# of CPU cores	Available memory (GB)	GPUs
Standard memory	524	48	180	
Medium memory	40	48	369	
Large memory	4	96	3014	
V100 NVIDIA GPU	11	48	369	4 32GB/16GB PCIe NVIDIA V100
RTX8000 NVIDIA GPU	73	48	369	4 48GB PCIe NVIDIA RTX8000
A100 NVIDIA GPU	9	64	500	4 80GB Nvlink NVIDIA A100
A100 NVIDIA GPU	34	80	1024	4 80GB Nvlink NVIDIA A100
AMD MI100 GPU	3	128	512	8 32GB PCIe AMD MI100
AMD MI250 GPU	2	128	512	8 64GB AMD MI250
NVIDIA H100 GPU	15	96	1510	4 80GB Nvlink NVIDIA H100

# Q/A



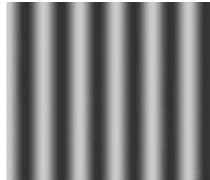
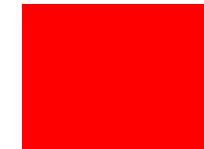
Smallest font



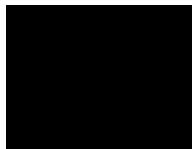
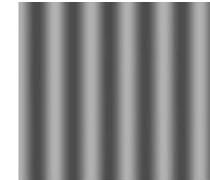
Please turn off and put  
away your cell phone



# Calibration slide



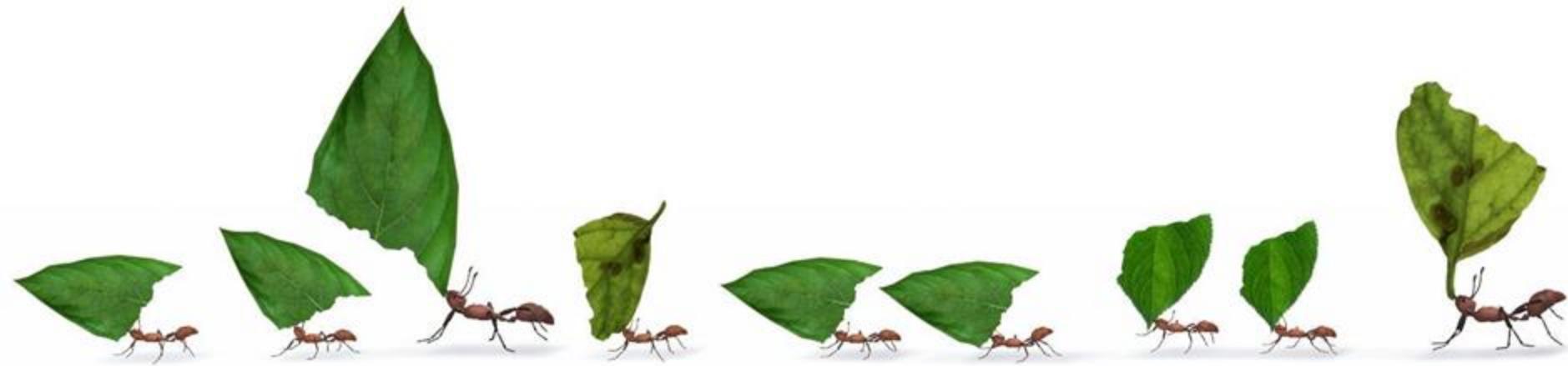
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



# This week

1. Finishing up Big Data infrastructure
2. Introducing Spark



NYU

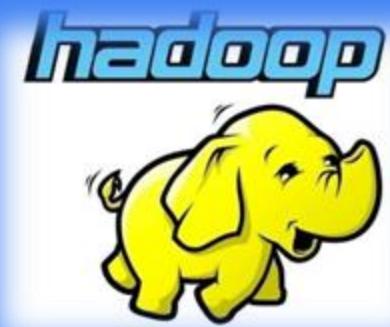
Center for  
Data Science

DS-GA 1004: Big Data

# 1) Big Data Infrastructure



# Taking a closer look at the Hadoop framework



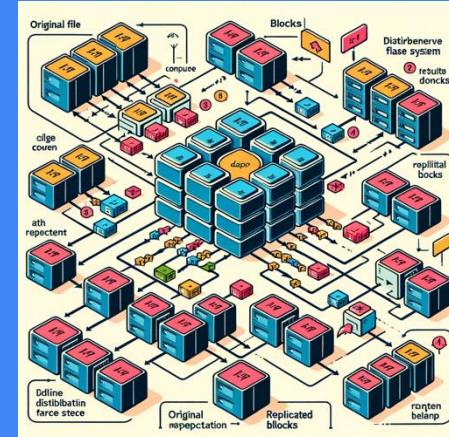
**MapReduce**

*Processing engine*



**YARN**

*Resource manager*



**HDFS**

*Storage layer*

# YARN was added to the Hadoop framework in version 2.0

Hadoop 1.x  
(until 2012)

MapReduce

HDFS

Hadoop 2.x  
(2012-2017)

MapReduce

Spark  
Flink  
Hive, Pig,...

YARN

HDFS

Hadoop 3.x  
(2017-now)

All  
engines  
from 2.x

Kubernetes  
Pods

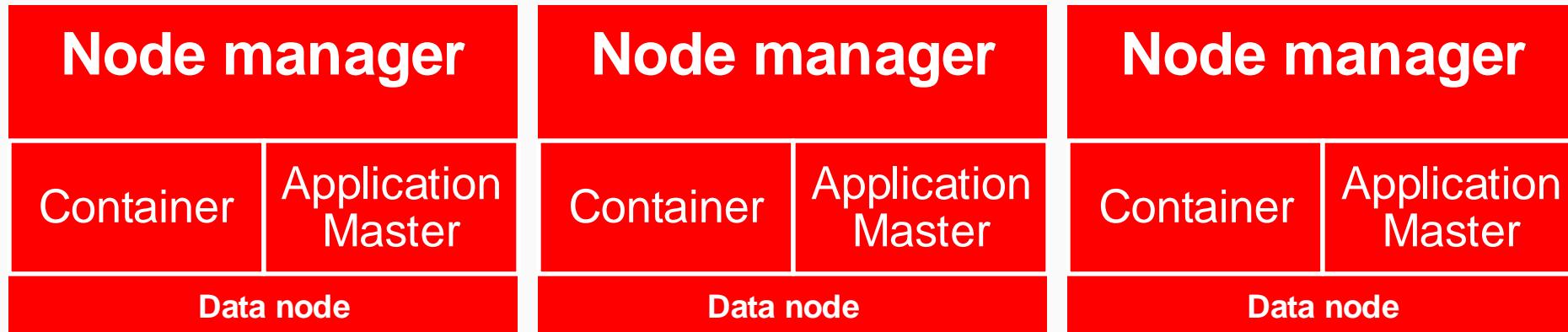
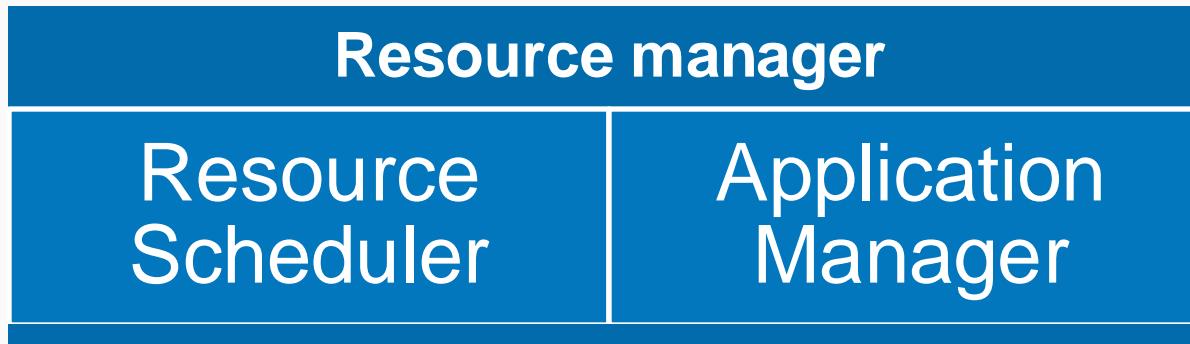
YARN

HDFS

Cloud  
storage

# YARN has effectively become the operating system of Hadoop

## **Yarn components:**



Actual footage of the interaction between resource manager and the application masters



# Cluster resources: Some terminology



Container

=



**NOT  
a docker  
container**

Storage space  
Processing cores  
Memory (RAM)

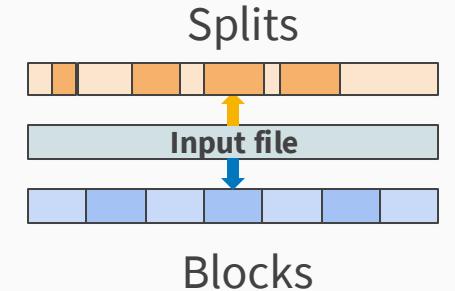
# Why should you care about implementation details?

Good example: The interplay between HDFS and Map-Reduce

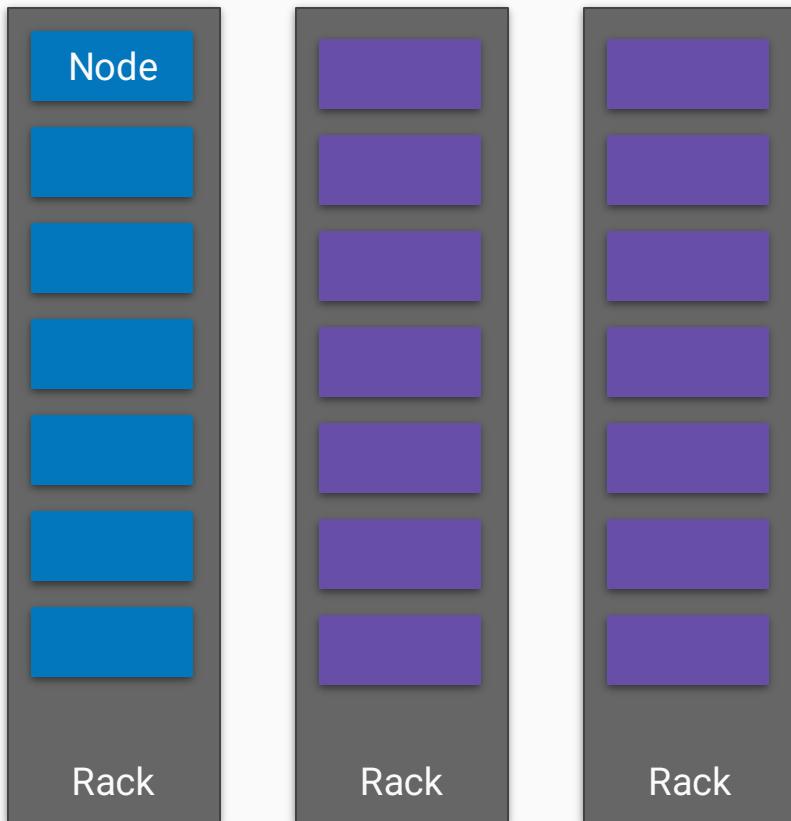
- HDFS shares **blocks** over data nodes
- Map-Reduce shares **jobs** over compute nodes
- What would be the case in an ideal world?
- If these were happening on the \*same\* node
  - For big data, bringing **compute ⇒ data** is cheaper than the other way around!

# So job scheduling and input splits can be coordinated / optimized

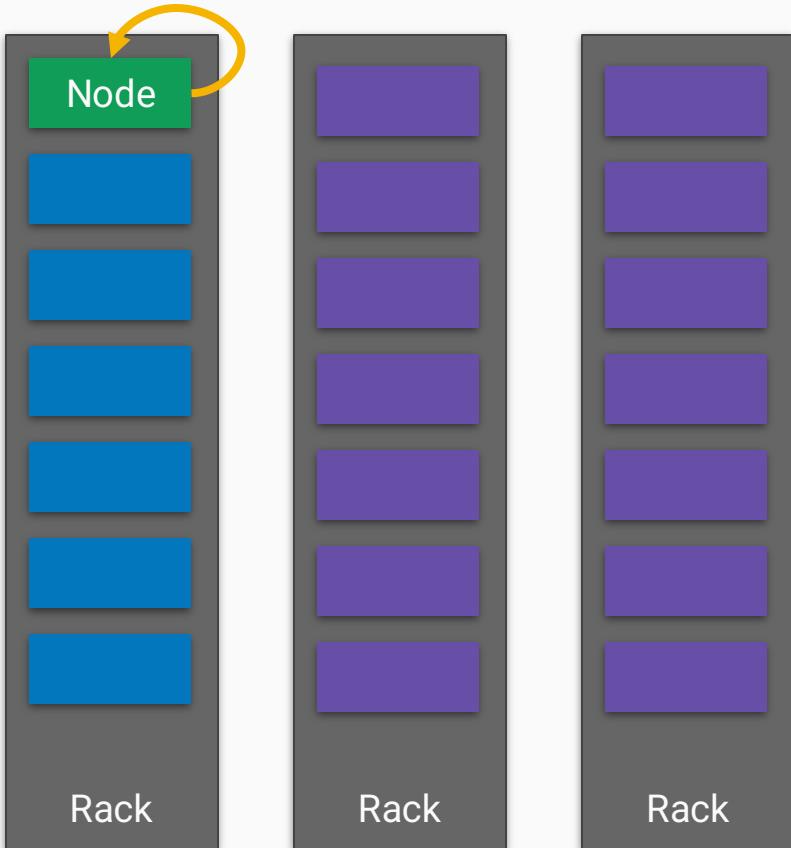
- A typical map-reduce job runs over one large data file
  - Each file contains a large number of (independent) records
- MapReduce divides the input into **splits**
- Each **split** maps onto one or more **blocks**
  - Optimal: Assign work such that processing of a **split** is done on a machine with its **blocks**
- HDFS exposes block layout to the job scheduler to make this possible



# Cluster organization: Network topology and interconnections between machines



# Where to execute a job?



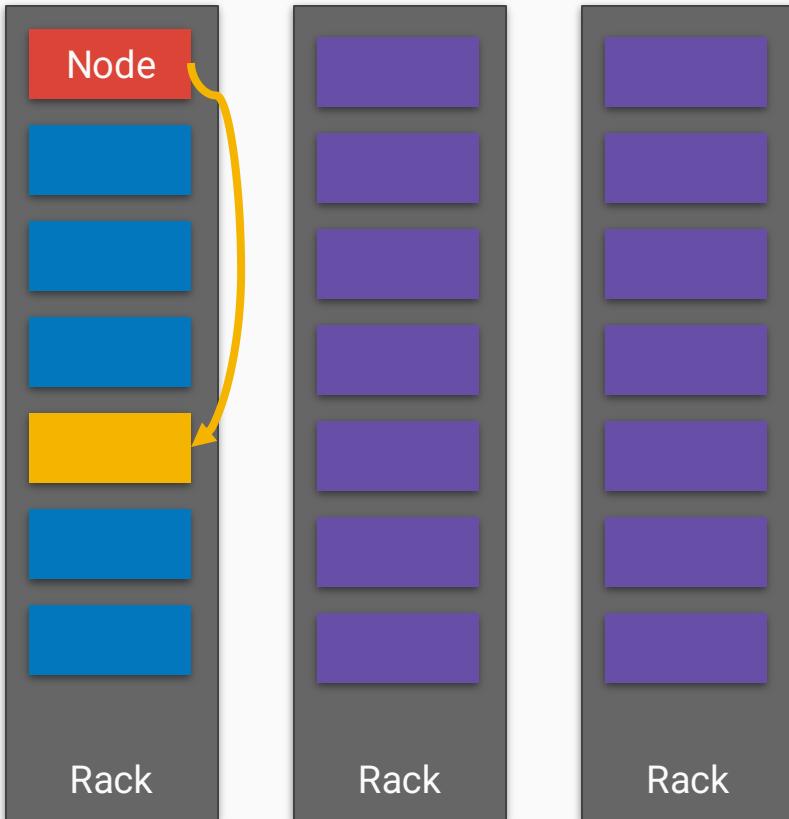
**Best case:**

Execute on a node that stores the block(s) we need

[https://commons.wikimedia.org/wiki/File:CERN\\_Server.jpg](https://commons.wikimedia.org/wiki/File:CERN_Server.jpg)

Florian Hirzinger – www.fh-ap.com, CC BY-SA 3.0 via Wikimedia Commons

# Where to execute a job?



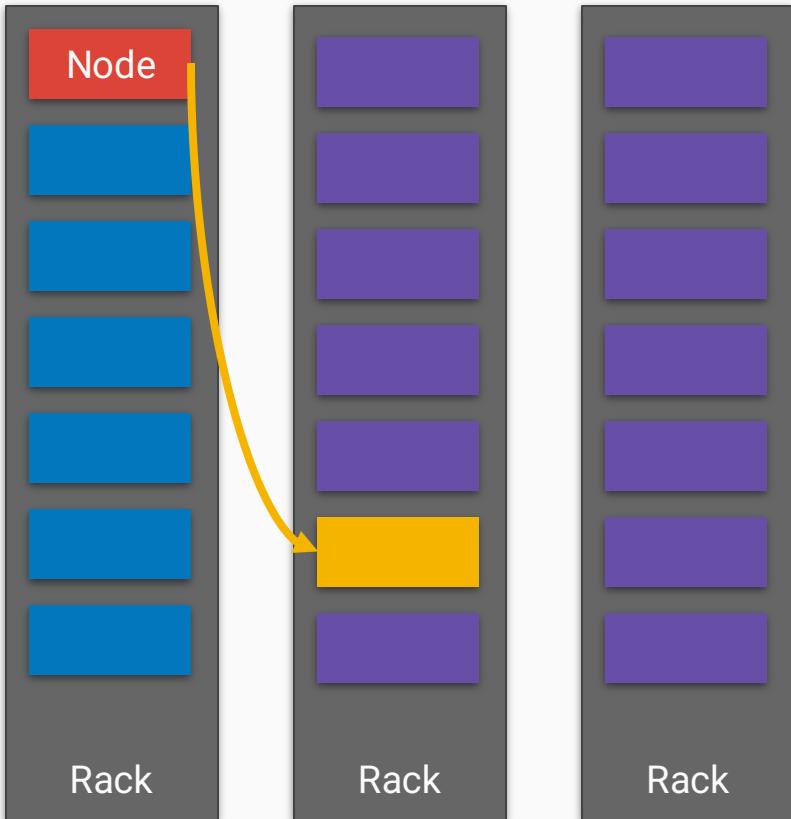
**Okay case:**

Execute on a different node in the same rack  
Within-rack communication is relatively fast

[https://commons.wikimedia.org/wiki/File:CERN\\_Server.jpg](https://commons.wikimedia.org/wiki/File:CERN_Server.jpg)

Florian Hirzinger – www.fh-ap.com, CC BY-SA 3.0 via Wikimedia Commons

# Where to execute a job?



**Worst case:**

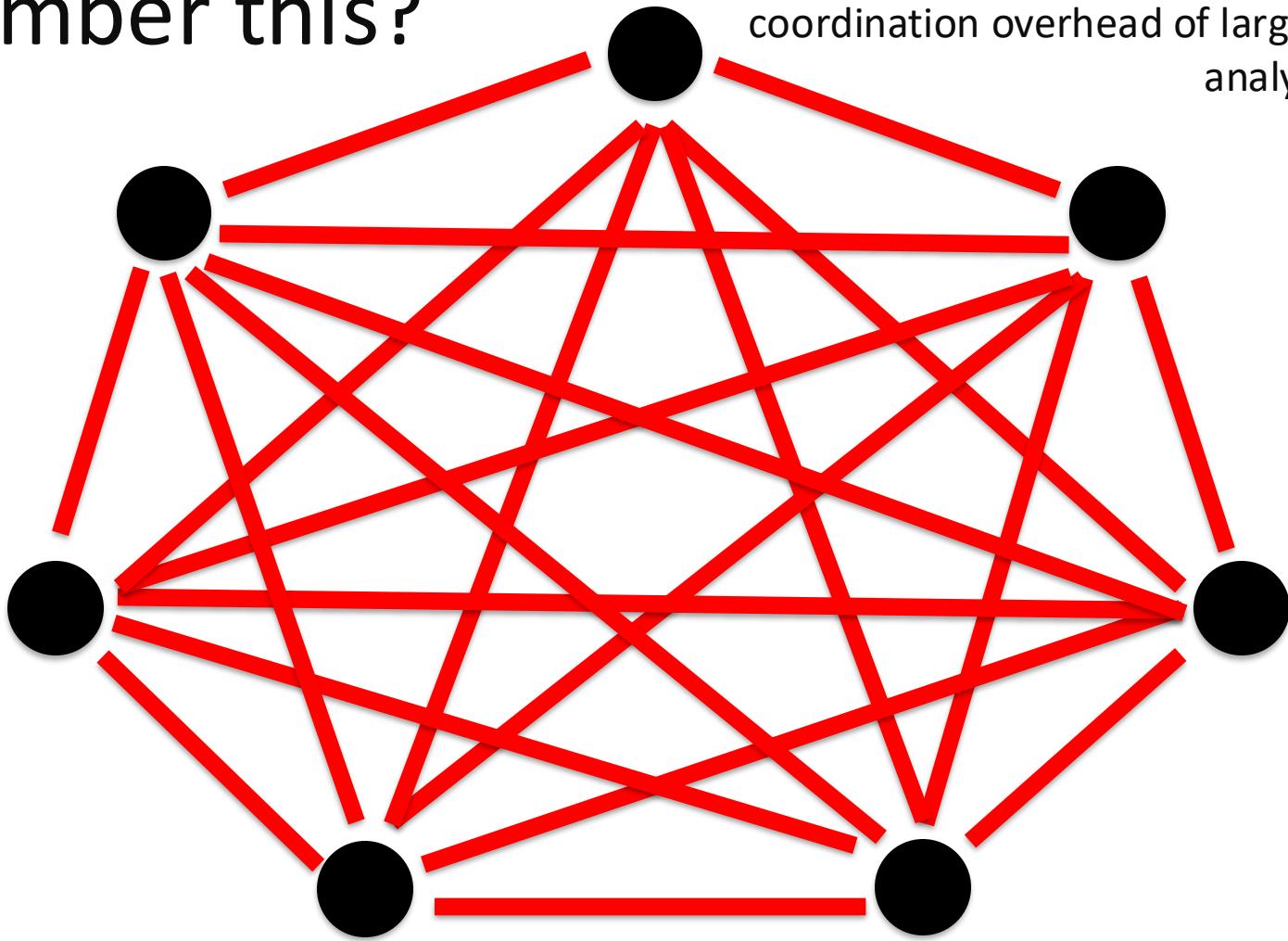
Execute on a node in a different rack  
Between-rack communication is slow

# Tied in with jobs: Replication factors

- Distribution of blocks to data nodes is not random.
- If we copy a block to multiple nodes, scheduling becomes easier
  - We're more likely to find a free worker that has the data we need for a given job
- HDFS lets you set the replication factor for each file
  - Replication isn't free: cost is a multiple of data size
- Default setup: 3x replication
  - If possible, 2 nodes in one rack, +1 in a separate rack
  - This protects against both **node failure** and **rack failure**

# Remember this?

It's about the necessary communication / coordination overhead of large scale (data analysis) projects



# The name of the game: Parallelizing large and complex data analysis tasks

- Everything we have done so far was in the service of doing so.
- Mostly by imposing restrictions.
- Restricting valid data structures (e.g. schemas)
- Restricting valid processing operations (e.g. MapReduce)
- Restricting storage modes (e.g. HDFS)

Is this too restrictive?

2) Spark



● mapreduce  
Search term

● Apache Spark  
Search term

● hdfs  
Search term

+ Add comparison

Worldwide ▾

2004 - present ▾

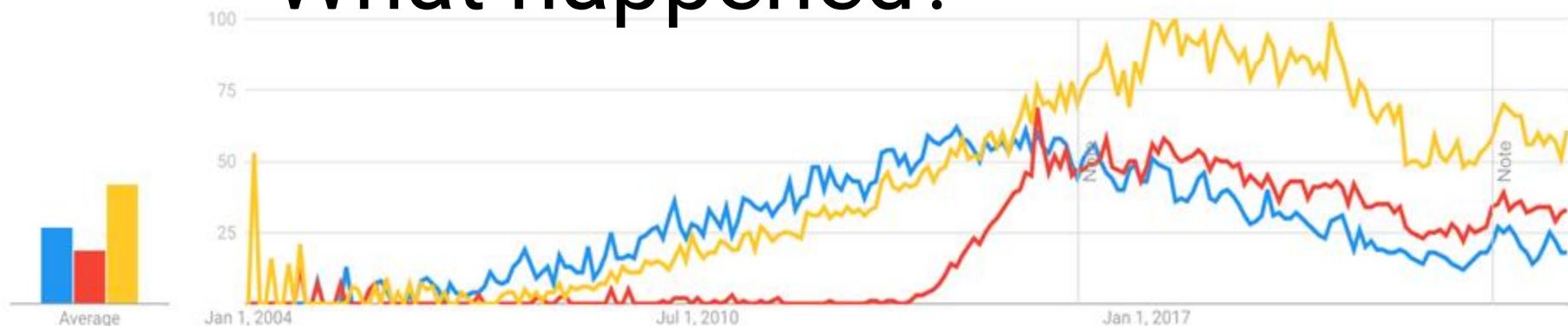
Computers & Electronics ▾

Web Search ▾

Interest over time ?



# What happened?





# What do they think about mapReduce and why did they create Spark?

Good:

- Scalability (allowing for parallel processing of “big data”)
- Fault tolerance
- So it works with off-the-shelf hardware (reliable despite unreliable components)
- Takes care of most of the “plumbing” (e.g. scheduling, load-balancing)

The key issue

- The “Stonebraker” criticism notwithstanding (“it’s a bad database”):
- mapReduce is built on an “acyclic data flow model”

# What do they mean by that? Is Map-Reduce... too low-level?

- Map-Reduce is great for one-time jobs with simple dependencies, just on big data. Fine for search, not for Data Science / Machine Learning:
- What if you want interactive or iterative procedures?
  - **Data exploration (EDA)**
  - **Complex queries** with multiple joins and aggregations
  - **Optimization** and machine learning

# Reminder from IDS: Gradient descent algorithm, on extremely \*small\* data

x	y
2	10

x: Cans of red bull

y: Price in \$

$\beta$ : Price per can

$$a_0 = 18 \quad C_0 = 64$$

$$C = (18-10)^2$$

$$a = x w$$

$$IR = 0.1$$

$$C = (a - y)^2$$

$w_0 = 9$  (initialized randomly)

$$\frac{\partial C}{\partial w} = \frac{\partial a}{\partial w} \frac{\partial C}{\partial a}$$

$$\frac{\partial C}{\partial a} = 2(a - y) = 2a - 20 = 2(xw) - 20 = 4w - 20$$

$$\frac{\partial a}{\partial w} = x = 2 \quad \frac{\partial C}{\partial w} = 8w - 40$$

Gradient descent  
is iterative:

$$w_1 = w_0 - (IR \frac{\partial C}{\partial w})$$

$$w_1 = 9 - (0.1 * (8*9 - 40)) = 5.8$$

$$a_1 = 11.6 \quad C_1 = 2.56$$

$$w_2 = 5.8 - (0.1 * (8*5.8 - 40)) = 5.16$$

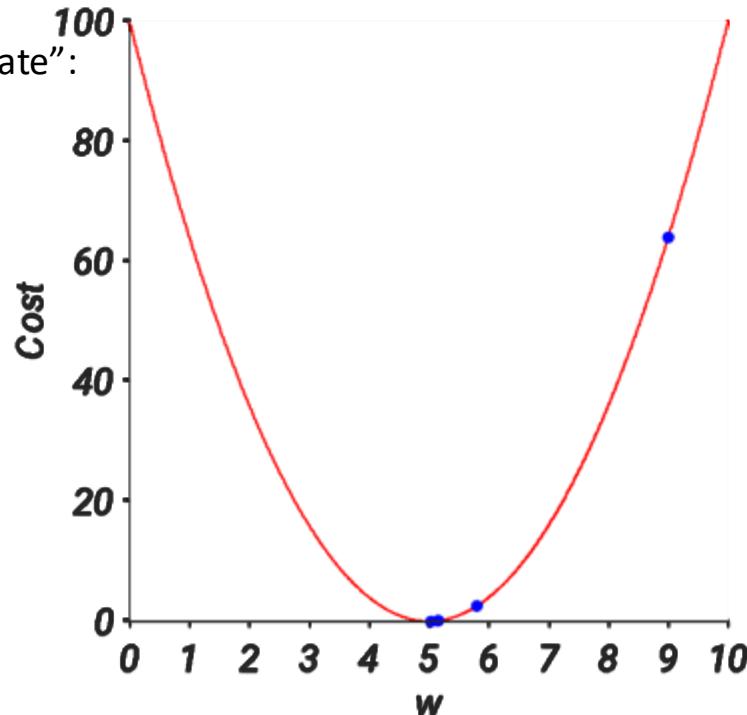
$$a_2 = 10.32 \quad C_2 = 0.1$$

$$w_3 = 5.16 - (0.1 * (8*5.16 - 40)) = 5.03$$

$$a_3 = 10.06 \quad C_3 = 0.004$$

For terminological reasons,  $\hat{y}$  is called “a” and  $\beta$  is typically called “w” in this framework (because it is often used in the context of neural networks)

Step size = “learning rate”:



$w \approx 5.03$ , with some extra steps

# Imagine implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n; w)$
- Initialize  $w$

# Imagine implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n; w)$
- Initialize  $w$
- Repeat until convergence:
  - mapper:  $x_n \rightarrow g_n = \nabla_w f(x_n; w)$  // N map jobs, compute gradients  
emit (1,  $g_n$ )

# Imagine implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n; w)$
- Initialize  $w$
- Repeat until convergence:
  - mapper:  $x_n \rightarrow g_n = \nabla_w f(x_n; w)$  // N map jobs, compute gradients  
emit  $(1, g_n)$
  - reducer:  $\{(1, g_n)\} \rightarrow G = \sum_n g_n$  // 1 reduce job, accumulate gradients  
emit  $G$
  - $w \leftarrow w - G$

# Imagine implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n ; w)$
- Initialize  $w$
- Repeat until convergence:
  - mapper:  $x_n \rightarrow g_n = \nabla_w f(x_n ; w)$  // N map jobs, compute gradients  
emit  $(1, g_n)$
  - reducer:  $\{(1, g_n)\} \rightarrow G = \sum_n g_n$  // 1 reduce job, accumulate gradients  
emit  $G$
  - $w \leftarrow w - G$

Each gradient step involves a full map-reduce!

And we don't even care about the previous iterations after they're done...

# Imagine implementing gradient descent on Big Data with MapReduce

- $\min_w \sum_n f(x_n ; w)$
- Initialize  $w$
- Repeat until convergence:
  - mapper:  $x_n \rightarrow g_n = \nabla_w f(x_n ; w)$  // N map jobs, compute gradients  
emit  $(1, g_n)$
  - reducer:  $\{(1, g_n)\} \rightarrow G = \sum_n g_n$  // 1 reduce job, accumulate  
emit  $G$
  - $w \leftarrow w - G$

Each gradient step involves a full map-reduce!

And we don't even care about the previous iterations after they're done...

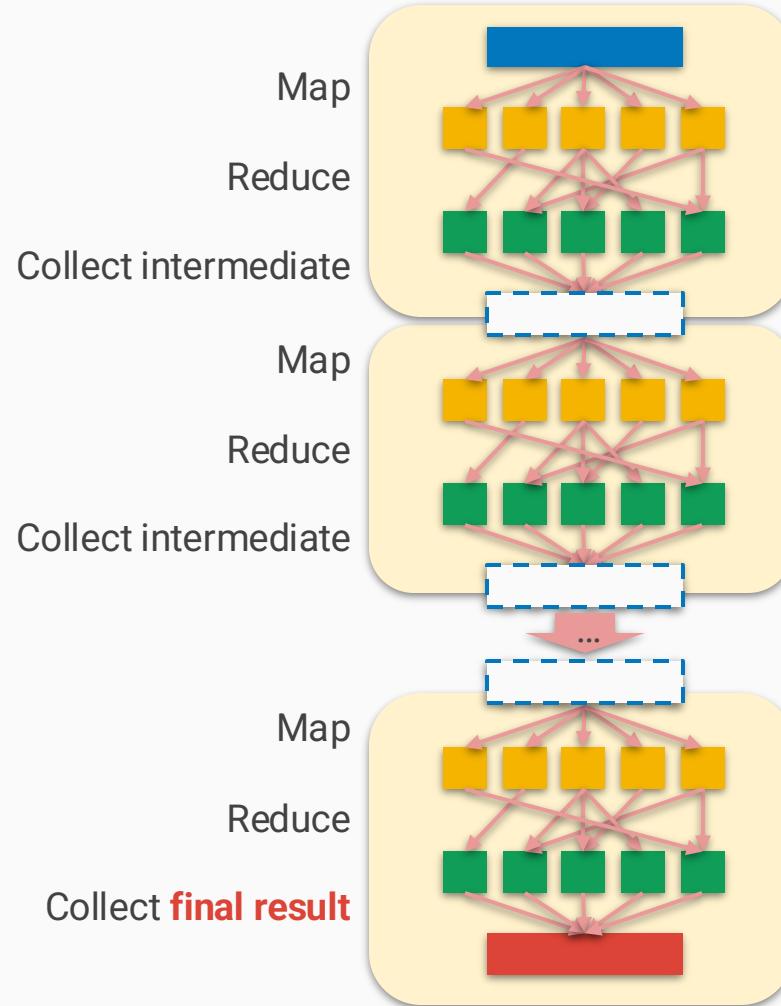
Reducer can't start until all mappers have finished  
⇒ high latency

# Complex pipelines

Computations can be decomposed into a sequence of MapReduce jobs

But this isn't always the easiest or most natural way to do it!

What if you want to rapidly iterate?



Upshot: Many/most commonly used machine learning methods rely on iterative algorithms (e.g. maximum likelihood estimation, gradient descent, kMeans, E/M algorithm, etc.)

Whereas it is possible to implement these with mapReduce, it is clunky and slow:

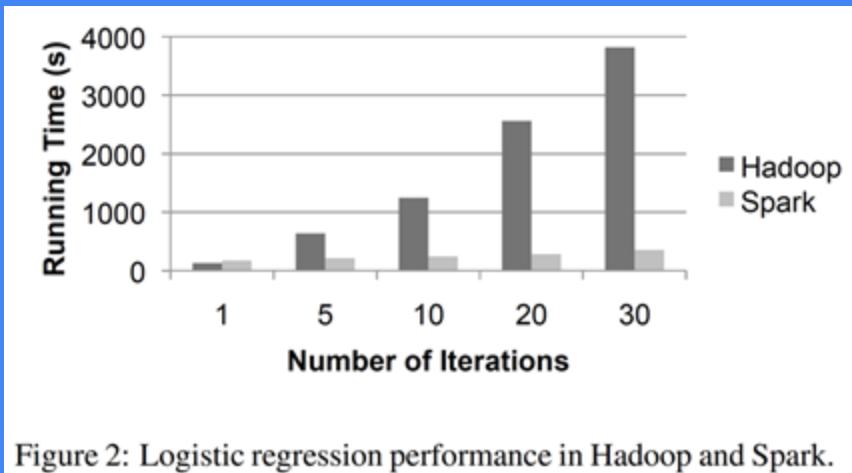


Figure 2: Logistic regression performance in Hadoop and Spark.

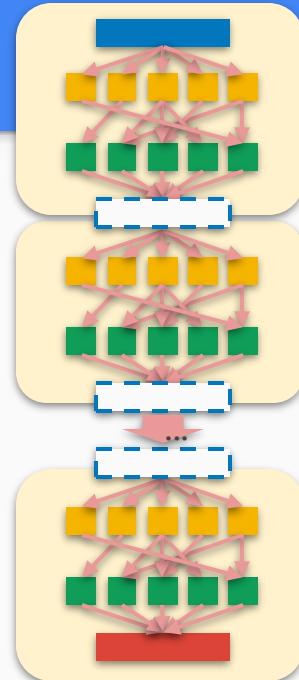
10x+ speedup  
for logistic  
regression

The proposed solution rests on a more flexible data structure:

Resilient distributed datasets  
(RDDs)

# The key idea: Reusing data

- Complex computations usually have many **intermediate steps**
- Map-Reduce paradigm favors the following pattern:
  - Compute each step
  - Store intermediate results
  - Move on to the next step
- This can be **wasteful** and **awkward** to implement



# Resilient distributed datasets (RDDs)

- RDD:
  - **Data source**
  - Lineage graph of **transformations** to apply to **data**
  - + interfaces for data **partitioning** and **iteration**
- A key concept: **Deferred computation**
  - Nothing is **computed** until you ask for it
  - Nothing is **saved** until you say so
  - This makes optimization possible

# Resilient distributed datasets (**RDDs**)

- RDD:
  - Linked to a **data source**
  - Lineage graph of **transformations** to apply to **data**
  - + interfaces for data **partitioning** and **iteration**
  - Immutable
- Think of this as **deferred computation**
  - Nothing is **computed** until you ask for it
  - Nothing is **saved** until you say so
  - This makes optimization possible

Some notation:

$\text{RDD}[T]$  denotes an RDD with some data of type T, e.g.

- $\text{RDD}[\text{String}]$
- $\text{RDD}[\text{Tuple}(\text{String}, \text{Float})]$

# RDD components: Implementing deferred computation

- **Transformations:** Operations on RDDs that return a new RDD.  
They are “**lazy**” (not executed immediately), but the computational steps are recorded in a **lineage graph**. Allows to efficiently create complex data processing pipelines.
- Examples: *map, filter, join*
- **Actions:** Trigger computation and yield results.
- Examples: *count, collect, reduce, take, save*

# RDD example: log processing

## Spark code

```
lines = spark.textFile("hdfs://...")
```

```
errors = lines.filter(_.startsWith("ERROR"))
```

```
errors.filter(_.contains("MySQL"))
    .map(_.split('\t')(3))
    .collect()
```

A math professor, a lumberjack, and a sysadmin are on a camping trip. They're sitting around the campfire when the sysadmin says, "Hey, let's take a look at the log and see what's been going on with the system."

The math professor responds, "Ah, you mean like the logarithmic function? That's a fascinating topic!"

The lumberjack chimes in, "No, no, I think he means the logs I've been cutting down. We could use them to keep the fire going."

The sysadmin shakes his head and says, "No, I mean the system log files. We can use them to troubleshoot any issues with the computer system."

The math professor looks at the lumberjack and says, "Well, I suppose we could use the logs to represent the logarithmic function in a visual way."

The lumberjack looks at the math professor and says, "Wait, what? Logs and logarithms are the same thing?"

The sysadmin laughs and says, "No, no, they're not the same thing at all. I just want to take a look at the log files on the computer system."

The math professor, lumberjack, and sysadmin all look at each other, realizing that they've been talking about completely different things.

The lumberjack shrugs and says, "Well, at least we've got plenty of real logs to keep the fire going!"

Legend:	Data	RDD	
	Transformation	Action	

# RDD example: log processing

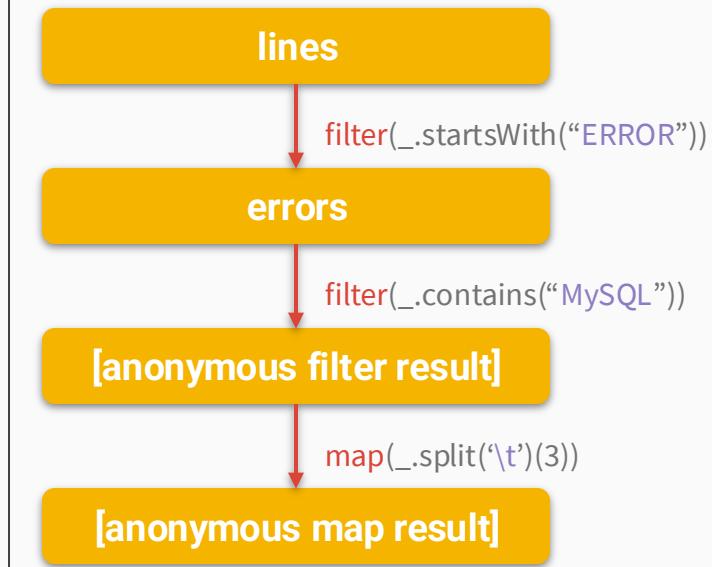
Spark code

```
lines = spark.textFile("hdfs://...")  
  
errors = lines.filter(_.startsWith("ERROR"))  
  
errors.filter(_.contains("MySQL"))  
    .map(_.split('\t')(3))  
    .collect()
```

No computation happens  
until you take an **action!**

Legend:    Data              RDD  
             Transformation      Action

Lineage graph



Adapted from [Zaharia et al., 2012]

# Transformations

Transformations turn one or more RDDs into a new RDD

Transformations are cheap to construct because they don't actually do the computation

Building an RDD is like **writing** (not *running*) a map-reduce script or a SQL query

- Examples:

- **map(function  $T \rightarrow U$ )**  $\Rightarrow \text{RDD}[T] \rightarrow \text{RDD}[U]$
- **filter(function  $T \rightarrow \text{Boolean}$ )**  $\Rightarrow \text{RDD}[T] \rightarrow \text{RDD}[T]$
- **union()**  $\Rightarrow (\text{RDD}[T], \text{RDD}[T]) \rightarrow \text{RDD}[T]$

```
lines = spark.textFile("hdfs://...")
```

```
errors = lines.filter(_.startsWith("ERROR"))
```

```
errors.filter(_.contains("MySQL"))
.map(_.split('\t')(3))
.collect()
```

# Actions

Actions are what execute the computations defined by an RDD

Results of actions are \*not\* RDDs

- Examples:
  - `count()` ⇒  $\text{RDD}[\text{T}] \rightarrow \text{Integer}$
  - `collect()` ⇒  $\text{RDD}[\text{T}] \rightarrow \text{Sequence}[\text{T}]$
  - `reduce(function (\text{T}, \text{T}) \rightarrow \text{T})` ⇒  $\text{RDD}[\text{T}] \rightarrow \text{T}$
  - `save(path)` ⇒ Save RDD to file system or HDFS

```
lines = spark.textFile("hdfs://...")
```

```
errors = lines.filter(_.startsWith("ERROR"))
```

```
errors.filter(_.contains("MySQL"))
    .map(_.split('\t')(3))
    .collect()
```

# Spark works backwards from actions towards the data source (through transformations)

```
lines = spark.textFile("hdfs://...")  
  
errors = lines.filter(_.startsWith("ERROR"))  
  
errors.filter(_.contains("MySQL"))  
    .map(_.split('\t')(3))  
    .collect()
```

1. **collect()** depends on **map()**
2. **map()** depends on **filter(MySQL)**
3. **filter(MySQL)** depends on **filter(ERROR)**
4. **filter(ERROR)** depends on **lines**
5. **lines** depends on **textfile**

# Spark works backwards from actions towards the data source (through transformations)

```
lines = spark.textFile("hdfs://...")  
  
errors = lines.filter(_.startsWith("ERROR"))  
  
errors.filter(_.contains("MySQL"))  
    .map(_.split('\t')(3))  
    .collect()
```

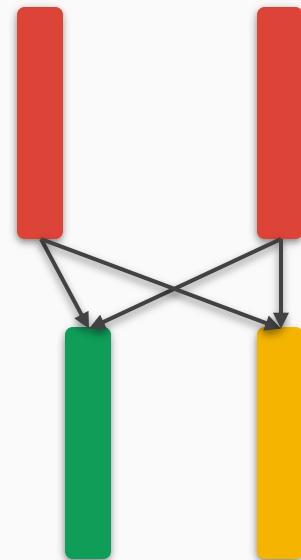
Any previously computed RDDs can be **cached** and reused!

Any lost / corrupted RDDs can be rebuilt from scratch by tracing the **lineage**!

1. **collect()** depends on **map()**
2. **map()** depends on **filter(MySQL)**
3. **filter(MySQL)** depends on **filter(ERROR)**
4. **filter(ERROR)** depends on **lines**
5. **lines** depends on **textfile**

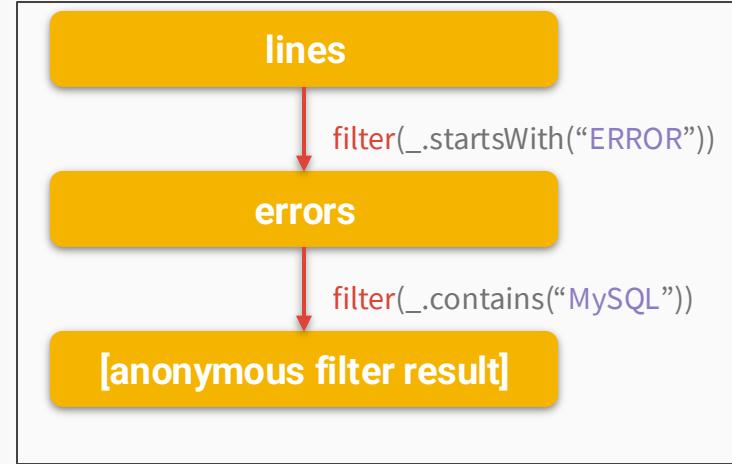
# The concept of a lineage graph

- It's called a "lineage graph", but it need not be linear!
- Any RDD can depend on multiple parent RDDs
- Once a parent RDD has been computed, it can be cached and reused by multiple descendants!
- This ability to reuse RDDs is what makes Spark so efficient for iterative algorithms.



# Pipelines

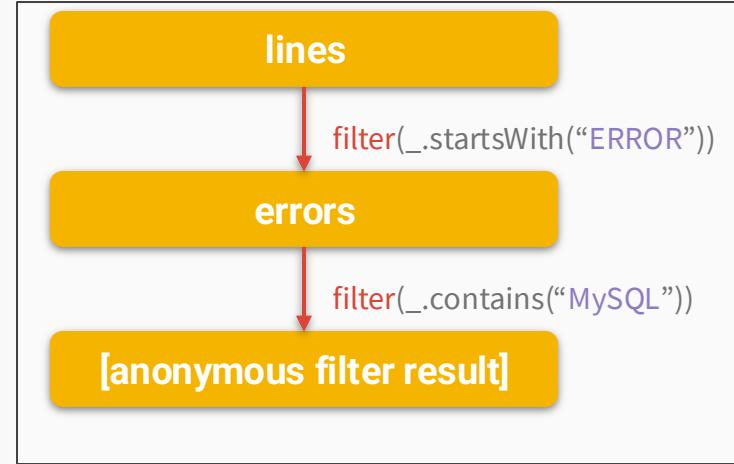
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch		

# Pipelines

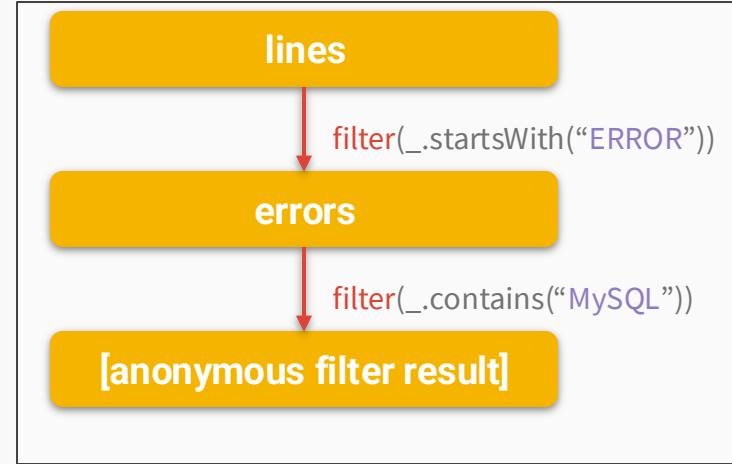
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK <b>Status OK</b> ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch		

# Pipelines

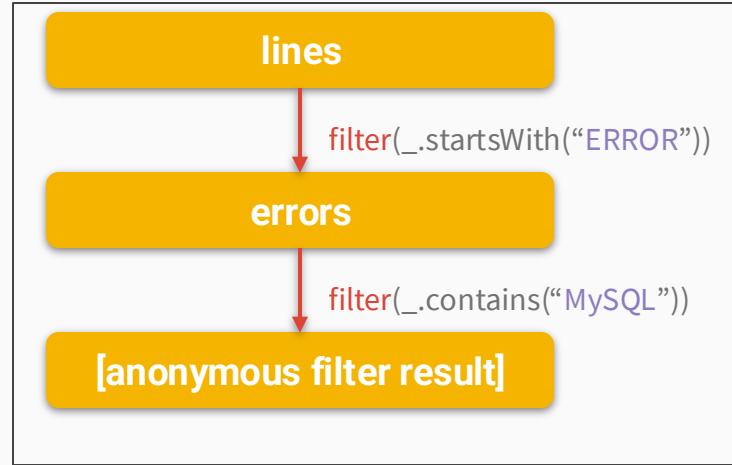
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK <b>ERROR: Rampaging T-Rex</b> Status OK ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex	

# Pipelines

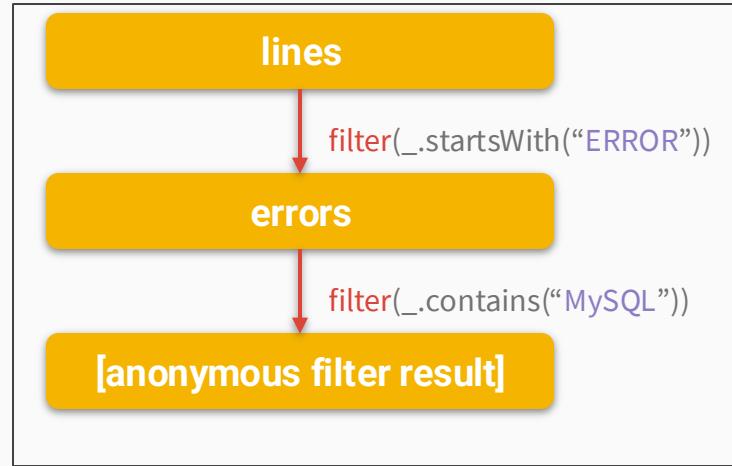
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex <b>Status OK</b> ERROR: MySQL failure Status OK ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex	

# Pipelines

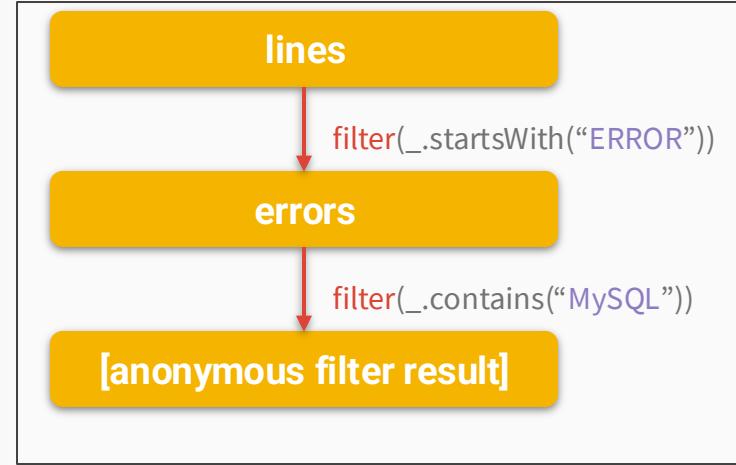
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK <b>ERROR: MySQL failure</b> Status OK ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex <b>ERROR: MySQL failure</b>	ERROR: MySQL failure

# Pipelines

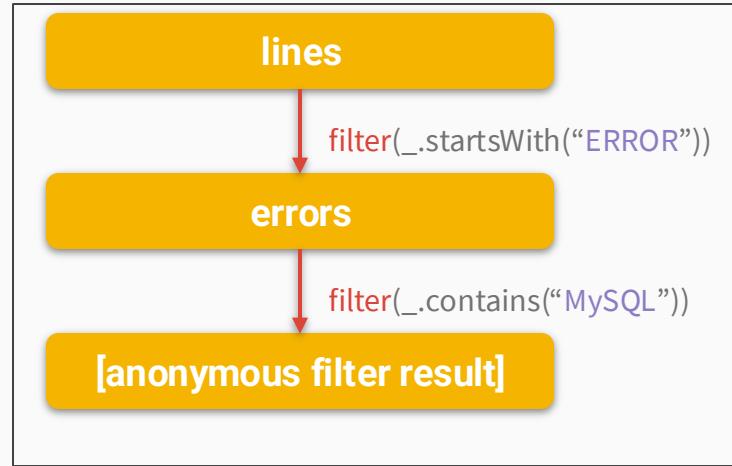
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure <b>Status OK</b> ERROR: Utahraptor ate my lunch	ERROR: Rampaging T-Rex ERROR: MySQL failure	ERROR: MySQL failure

# Pipelines

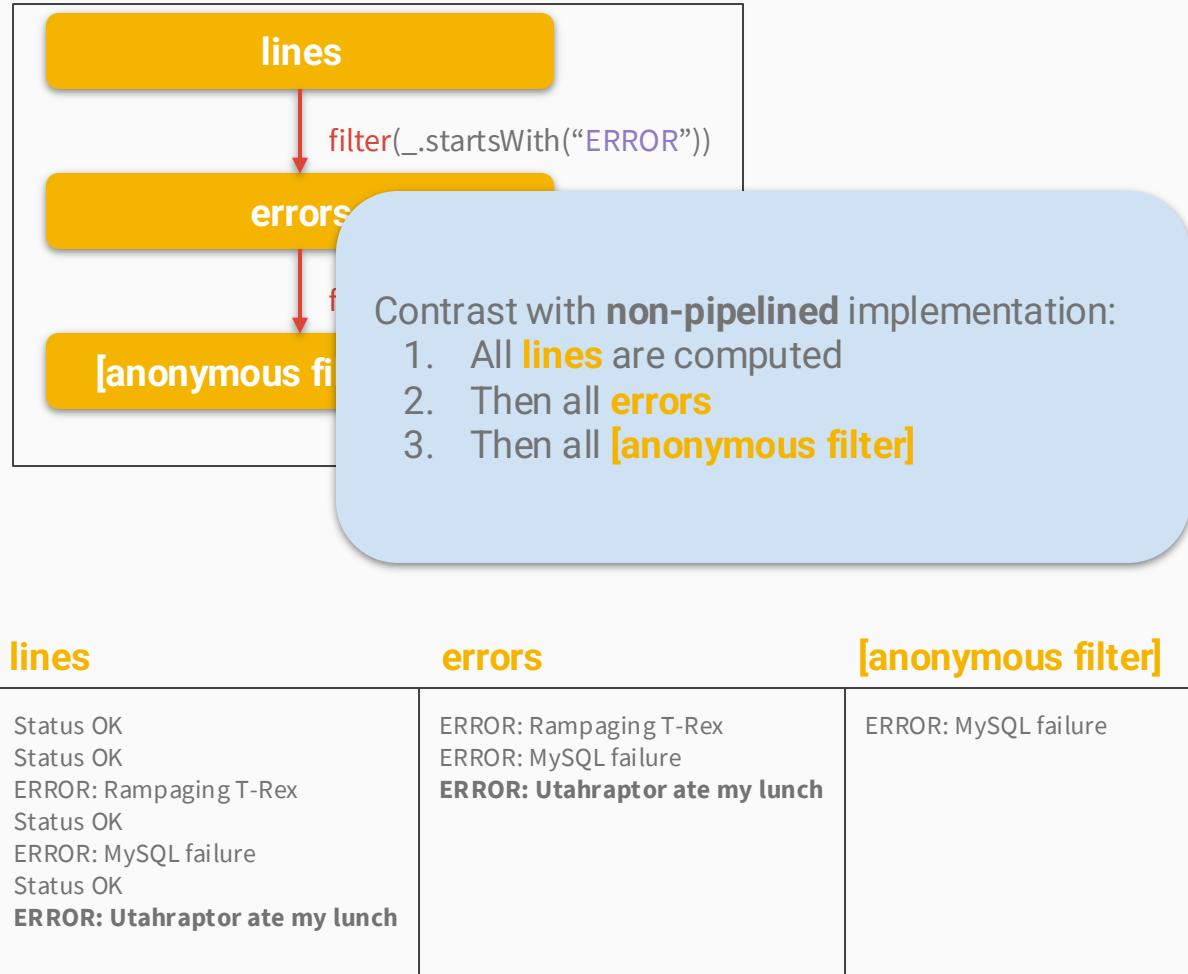
- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



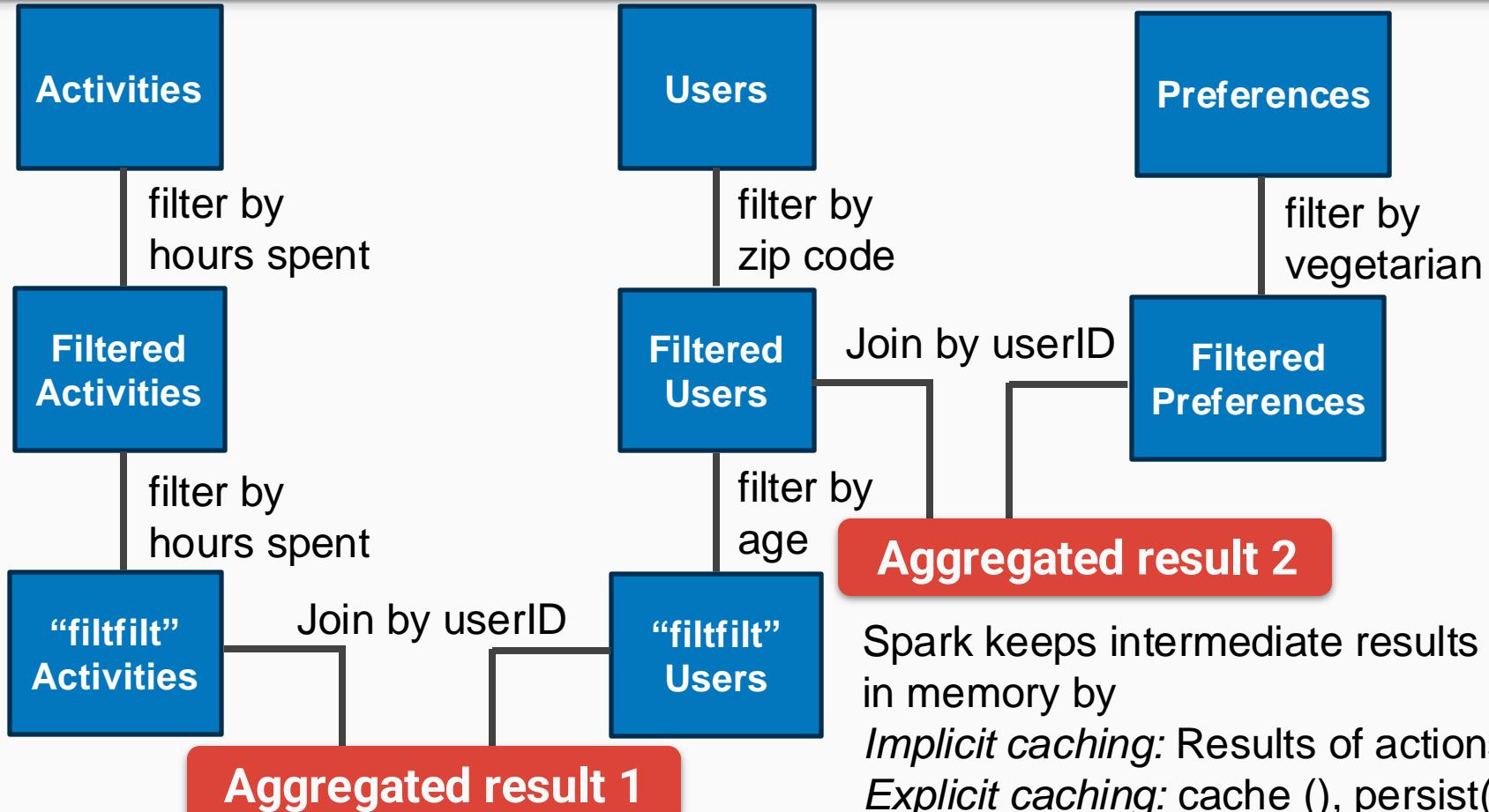
lines	errors	[anonymous filter]
Status OK Status OK ERROR: Rampaging T-Rex Status OK ERROR: MySQL failure Status OK <b>ERROR: Utahraptor ate my lunch</b>	ERROR: Rampaging T-Rex ERROR: MySQL failure <b>ERROR: Utahraptor ate my lunch</b>	ERROR: MySQL failure

# Pipelines

- Lineages can be **pipelined**
- We don't need to wait for all of **lines** to finish to build **errors**
- **No need for intermediate storage like in Map-Reduce**



# An example lineage graph of a multi-parent RDD pipeline

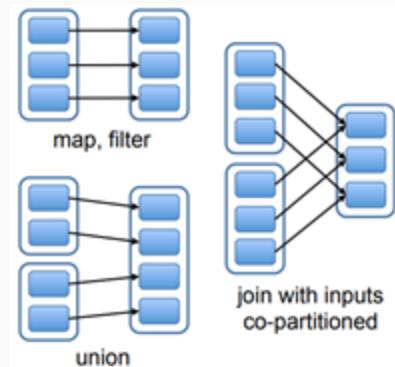


# Partitions: Narrow and wide dependencies

## Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery

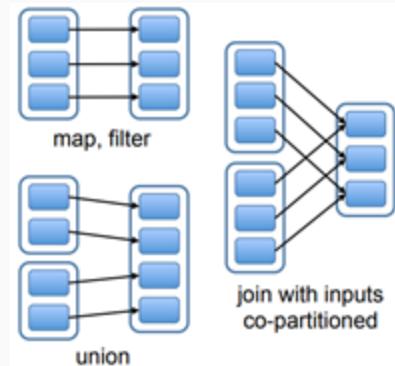


# Partitions: Narrow and wide dependencies

## Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

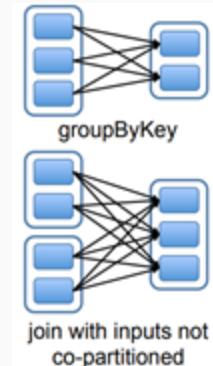
- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery



## Wide dependencies

Partition of parent RDD goes to multiple child RDD partitions

- High communication
- High latency
- Difficult to pipeline
- Difficult to recover



# Example: RDDs and pipelines in Spark



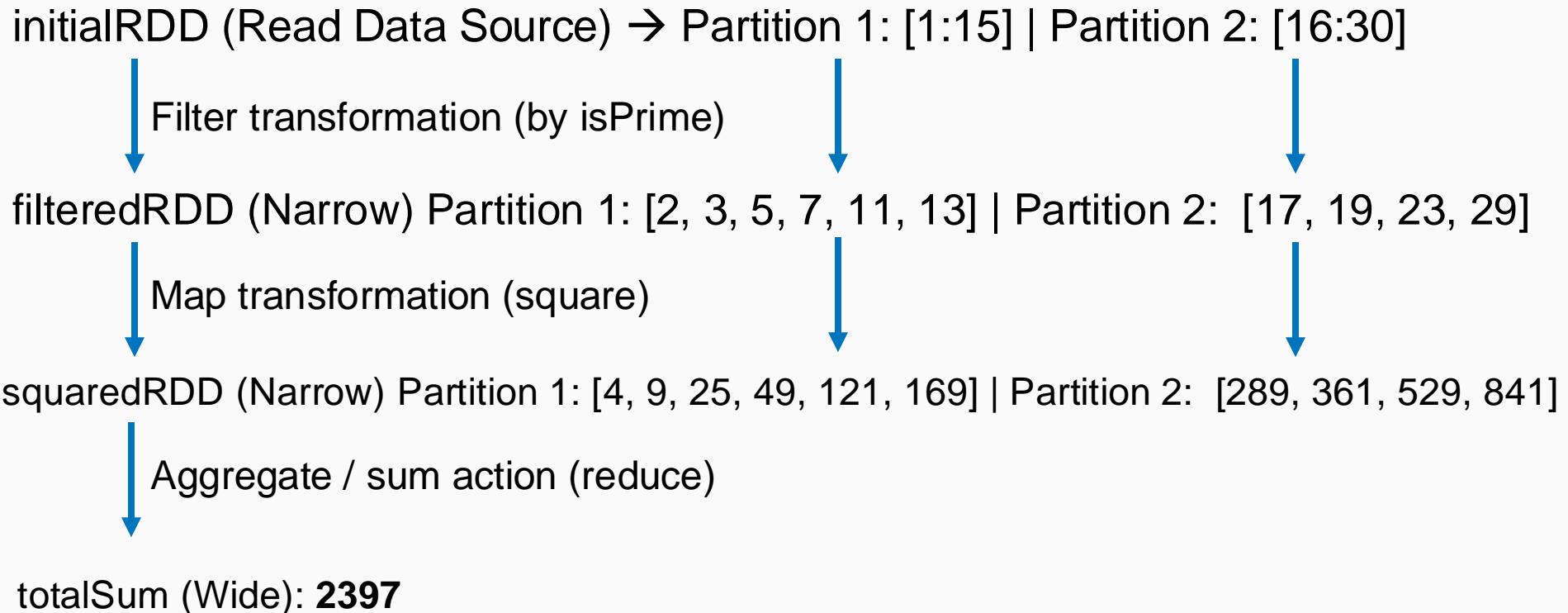
initialRDD: Creates partitions by using, e.g. *parallelize* on the data source

filteredRDD: Filtering the initialRDD, e.g. by *filter* primes,  
a narrow dependency transformation

squaredRDD: Squaring the filteredRDD, e.g. by *map*,  
a narrow dependency transformation

totalSum: Aggregating the sum by applying *reduce* to the squaredRDD,  
a wide dependency action

# Example: RDDs & Partitions



**Caution:** Much like “bias” in machine learning, “partition” seems to be the favorite word in Big Data, with many different meanings

- In **Hadoop / CAP theorem**: Network partition, disconnected nodes in a network.
- In **Spark**: Data partition, a chunk of data
- For **disks**: Logical division of a hard drive into storage sections
- For **databases**: Distributing large databases into chunks across nodes (sharding, e.g. MongoDB)
- ...

# RDDs

- Resilient Distributed Datasets (RDDs) are the fundamental data structure of Spark
- Spark uses deferred computation to efficiently construct complex analyses
  - Transformations vs actions!
- RDD partitions are analogous to map-reduce splits, and allow parallel execution

# Next week

- Applied Spark
- Column-oriented storage (Parquet)
- Dremel

**Q & R**

**spark**



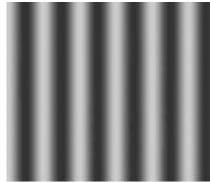
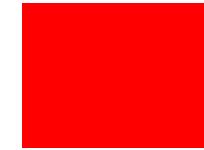
Smallest font



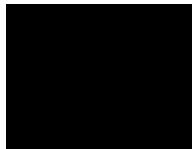
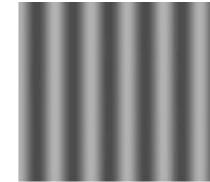
Please turn off and put  
away your cell phone



# Calibration slide



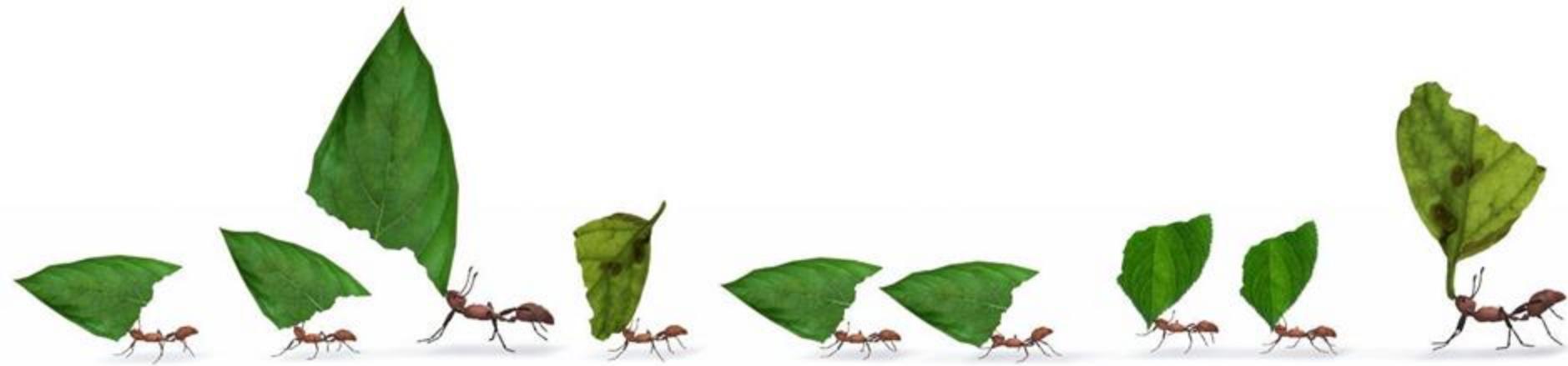
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



# Week 07:

# Column Oriented Storage

DS-GA 1004: Big Data

# Today

- CDS
- Using Spark (today)
- **Column-oriented storage**
- Dremel and Parquet

**TLDR:** parallelism isn't everything.

Data structures are still important!

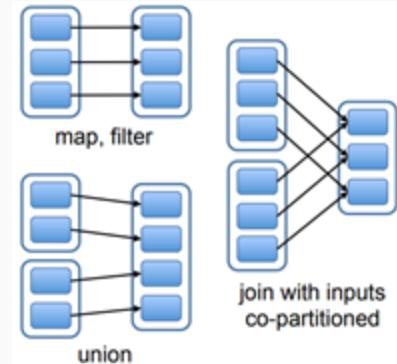
# CDS

## Partitions: Narrow and wide dependencies

### Narrow dependencies

Partition of parent RDD goes to at most 1 partition of child RDDs

- Low communication
- Localized
- Easy to pipeline
- Easy failure recovery



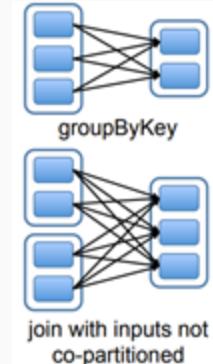
**No data needs to move: Fast**

### Wide dependencies

Partition of parent RDD goes to multiple child RDD partitions

- High communication
- High latency
- Difficult to pipeline
- Difficult to recover

**Key shuffle required: Slow**



Figures adapted from [Zaharia et al., 2012]

## Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT
1	Alex	3	1
2	Brett	1	0
3	Corey	2	0
4	Drew	2	0
5	Emory	1	1
6	Frankie	2	1

mID	Name	School
1	DS	CDS
2	CS	Courant
3	Math	CAS

**Students (s)**

**Majors (m)**

**Academic Records (ar)**

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

How to check if full time students take enough credits?

One way: Join S and AR by student id (sID)

However: Join is a wide dependency

It requires a key shuffle. This takes a lot of time.

## Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

Co-partitioning can turn wide dependencies into narrow ones by obviating the need for data to move.

The idea is that co-partitioning ensures that the necessary data is already on the right partition for future use.

Note: This future access pattern has to be anticipated (and specified) by you:

**df.repartition(# PARTITIONS, col("NAME OF COLUMN"))**

Here, for instance:

```
s.repartition(3,sID)  
ar.repartition(3,sID)
```

Actual co-partitioning happens by virtue of a hash function that deterministically allocates the same data partition to records with the same common key, e.g.:

1 → 1,    2 → 2,    3 → 3,    4 → 1,    5 → 2,    6 → 3

## Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT
1	Alex	3	1
2	Brett	1	0
3	Corey	2	0
4	Drew	2	0
5	Emory	1	1
6	Frankie	2	1

mID	Name	School
1	DS	CDS
2	CS	Courant
3	Math	CAS

**Students (s)**

**Majors (m)**

**Academic Records (ar)**

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

But doesn't co-partitioning just do a key shuffle up front?

This is true. So what is gained by doing it up front?

If "student" is a commonly used access pattern:

For instance, computing GPA.

One planned co-partitioning can avoid many ad-hoc key shuffles

# Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

sID	Name	Major	FT
1	Alex	3	1
2	Brett	1	0
3	Corey	2	0
4	Drew	2	0
5	Emory	1	1
6	Frankie	2	1

mID	Name	School
1	DS	CDS
2	CS	Courant
3	Math	CAS

**Students (s)**

**Majors (m)**

**Academic Records (ar)**

sID	Course	Grade	Credits
1	Analysis	C	4
1	IDS	B+	4
1	LA	B	3
2	IDS	B	4
3	IDS	A-	4
4	IDS	A-	4
4	ML	C+	3
5	IDS	C	4
5	Big Data	B	4
5	ML	A	3
6	Big Data	B-	4

However: This effect is very specific:

Only records co-partitioned with the common key benefit

For instance, counting the number of students per major

To do this efficiently, we would have to co-partition on mID:

s.repartition(3, Major)

m.repartition(3, mID)

## Confusion, Doubt & Struggle: The problem of wide dependencies/need for copartitioning

Some notes on co-partitioning:

Co-partitioning optimizes operations that use the same common key (only). So the benefits from co-partitioning are key-specific. The strongest benefits come from access patterns that often use a common key (like student id here).

There are different ways to co-partition:

“**repartition**” co-partitions in memory, ephemerally.

“**bucketBy**” co-partitions to disk memory, permanently (bucketing)

How does Spark know that co-partitioning happening and that the data is already partitioned properly, and no additional shuffle is needed?

Spark keeps track of how data is partitioned with metadata.

Spark has a “Catalyst” optimizer (written in Scala) that transforms high level queries into efficient execution plans.

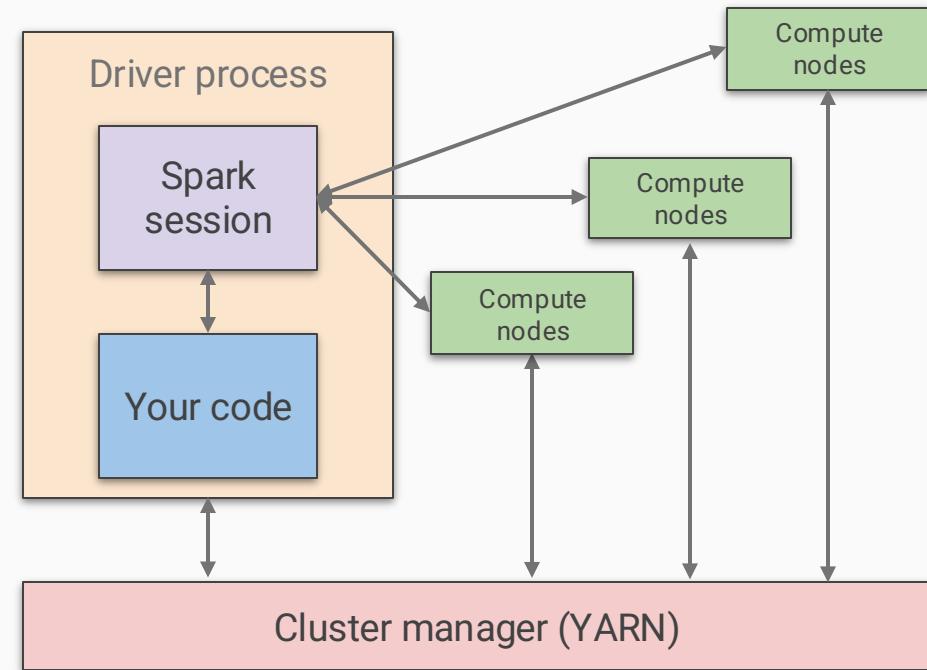
Using Spark \*now\*  
(it has come a long way)

# Apache Spark (developed in 2009, introduced 2010, published 2012, released 2013)

- Cluster computing framework using RDDs
- Integrates with Hadoop ecosystem
  - HDFS for storage (but other backends are possible)
  - (Hadoop) YARN for scheduling
- Written in Scala with APIs in other languages
  - Python (e.g. via PySpark), Java, R, MATLAB (MDCS), etc.

# Architecture: session and driver

- **Driver** is the process that you run on, e.g. the head/login node
- The **Session** object connects **your code** to the **cluster / compute nodes**



# Modern Spark: Beyond Scala

- You don't need to code Scala to use Spark
- Spark can run from R or Python (or Java)
- Beware: R and Python may not be as fast as Scala
- **Crossing process boundaries** can be expensive, but Spark does a good job of managing this

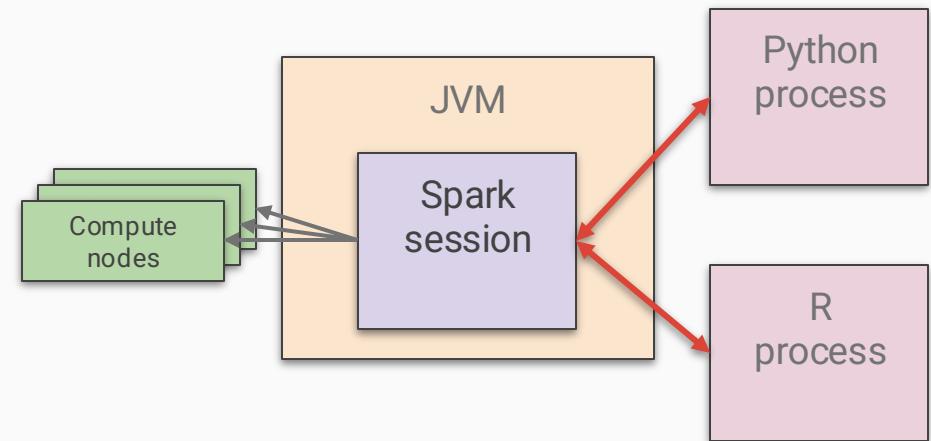


Figure adapted from [Chambers and Zaharia, 2018]

# Spark DataFrames API

- RDDs are great, but a bit cumbersome for ad-hoc computation
- **DataFrames** are common representations in many languages
  - R, pandas (Python), etc.
- Spark 2.x added a DataFrame API as a primary interface
  - Code looks more or less like pandas/Python!

# RDDs are more than columns

- RDDs can be derived from other RDDs through **transformations**
- RDDs also expose **partition** information, which influences how data is stored in HDFS
- Since Spark 2.0, one or more RDDs can be assembled to form DataFrames

# DataFrames and RDDs

- DataFrames in Spark are like relations in RDBMS
  - Well-defined schema with types over columns
  - Each row is a tuple (sort of...)
- DataFrame operations are translated into RDD transformations by Spark
- RDD transformations can then be executed within JVM
  - No more back and forth between JVM $\Leftrightarrow$ Python!

# DataFrames and RDDs

- When using RDBMS (and DataFrames) we often think of data in **rows**
  - 1 row = 1 record
  - ≈ 1 line in a CSV
- DataFrames are implemented as **collections of RDDs**
  - 1 column = 1 RDD
- As a user, this doesn't change much, but it does change how we think about **storage** ⇒ column-oriented storage (coming up)

# Spark-SQL

- Spark 2.x allows you to express queries in SQL
- Queries are executed against DataFrames
  - DataFrames are secretly RDDs, not RDBMS tables
- Queries can be optimized by analyzing the RDD lineage graph

```
df.createOrReplaceTempView('my_table')

res = spark.sql(' SELECT zip_code, sum(height) as H
                  FROM my_table
                 GROUP BY zip_code')
res.show()
```

---

```
df.groupBy('zip_code')
   .sum('height')
   .withColumnRenamed('sum(height)', 'H')
   .show()
```

# Tips and pitfalls

- Before running an action, run the `explain()` method on the DataFrame
  - This will give you an execution plan
  - You might identify some inefficiencies or bugs this way
- Be careful with `collect()`!
  - This will stream all results back to the driver node
  - If it's a large data set, and you forgot an aggregation step, this will be very bad news.
  - Test-drive a large query with `take(10)` instead of `collect()`
  - Probably you want `.save()` instead of `.collect()` anyway

中文的标准书写方向为从上到下。

Now

# Column-oriented storage

*"The English fight in a line, but the French fight in columns."*

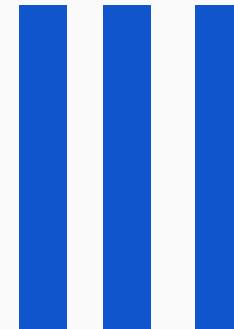
-Napoleon, 1815

(At Waterloo, supposedly)

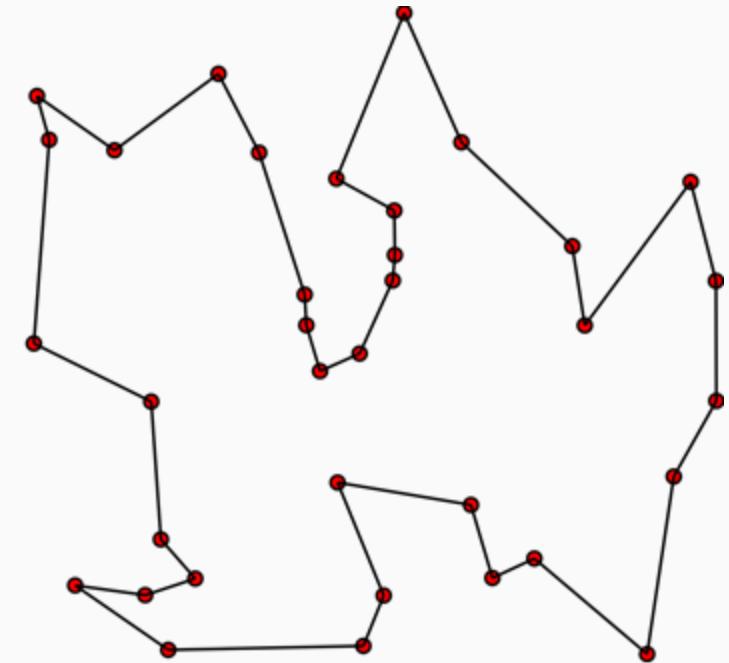
***The standard directionality in English is left to right.***



vs

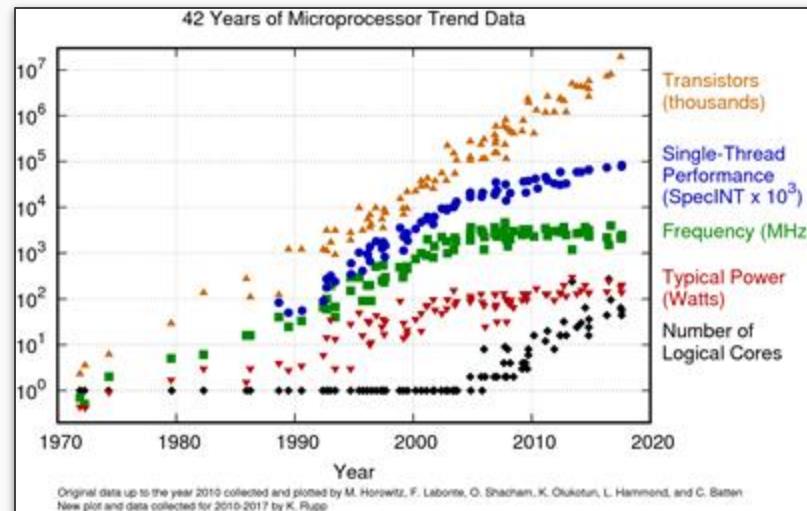


Heuristic: It's all about speed (time), as many big data processing jobs don't scale gently



# Column oriented storage history

- Idea goes back to the 1970s and 1980s
  - Transposed files [Batory, 1979]; Cantor [Karasalo & Svensson, 1983]
- Resurgence in the 2000s
  - MonetDB [Boncz & Kersten, 2002]
  - C-Store [Stonebraker et al., 2005]
  - VectorWise [Idreos et al., 2008]
- Why the resurgence?
  - Increased CPU speed + deep pipelining
  - Stagnant storage speed



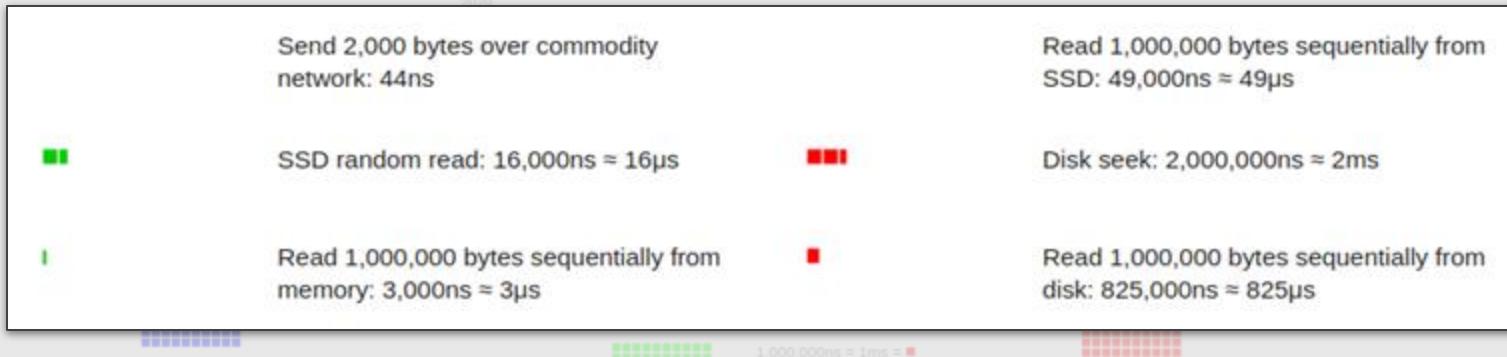
## Latency Numbers Every Programmer Should Know

2020

■	1ns	■	Main memory reference: 100ns	Send 2,000 bytes over commodity network: 44ns	Read 1,000,000 bytes sequentially from SSD: 49,000ns = 49μs
■	L1 cache reference: 1ns	███████	1,000ns = 1μs	SSD random read: 16,000ns = 16μs	Disk seek: 2,000,000ns = 2ms
■■■	Branch mispredict: 3ns	███████	Compress 1KB wth Zippy: 2,000ns ≈ 2μs	Read 1,000,000 bytes sequentially from memory: 3,000ns = 3μs	Read 1,000,000 bytes sequentially from disk: 825,000ns = 825μs
■■■■	L2 cache reference: 4ns	█████	10,000ns = 10μs = ■	Round trip in same datacenter: 500,000ns ≈ 500μs	Packet roundtrip CA to Netherlands: 150,000,000ns = 150ms
■■■■■	Mutex lock/unlock: 17ns	██████████		1,000,000ns = 1ms = ■	
	100ns = ■	██████████		██████████	██████████

## Latency Numbers Every Programmer Should Know

■	1ns
■	L1 cache reference: 1ns
■■■	Branch mispredict: 3ns
■■■■	L2 cache reference: 4ns
■■■■■■■■	Mutex lock/unlock: 17ns



- Transferring from disk to memory is incredibly slow
- Sequential memory reads are faster due to cache pre-fetching
- Strategies:
  - Transfer fewer bytes
  - Use predictable and contiguous memory access patterns

# Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```

# Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

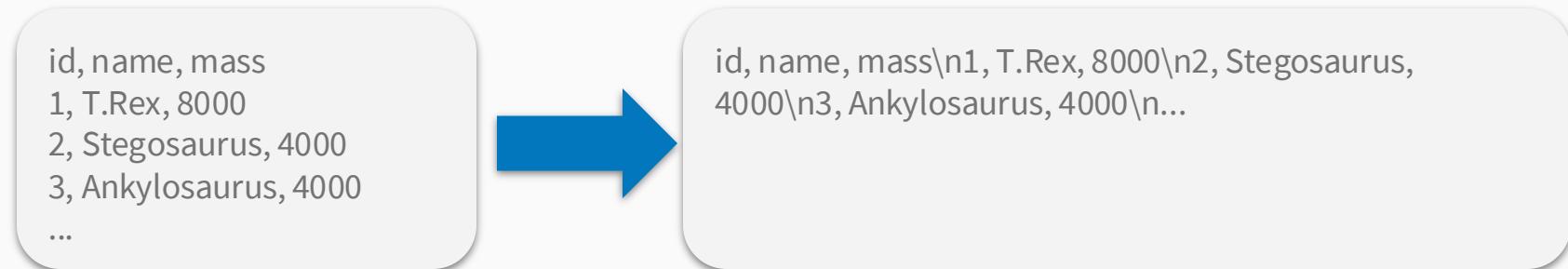
```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```



```
id, name, mass\n1, T.Rex, 8000\n2, Stegosaurus, 4000\n3, Ankylosaurus, 4000\n...
```

# Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way



- How would you access the 1000th record?
- How would you access just the third column?

# Row-oriented storage: CSV files

- Imagine you have data stored as rows of text in the usual way

```
id, name, mass  
1, T.Rex, 8000  
2, Stegosaurus, 4000  
3, Ankylosaurus, 4000  
...
```



```
id, name, mass\n1, T.Rex, 8000\n2, Stegosaurus, 4000\n3, Ankylosaurus, 4000\n...
```

## Problems:

- Records are variable-length
- Row and column offsets are hard to predict
- Basically requires full serial scan
- Full serial scan is slow ( $O(n)$ , technically  $O(n/2)$ )

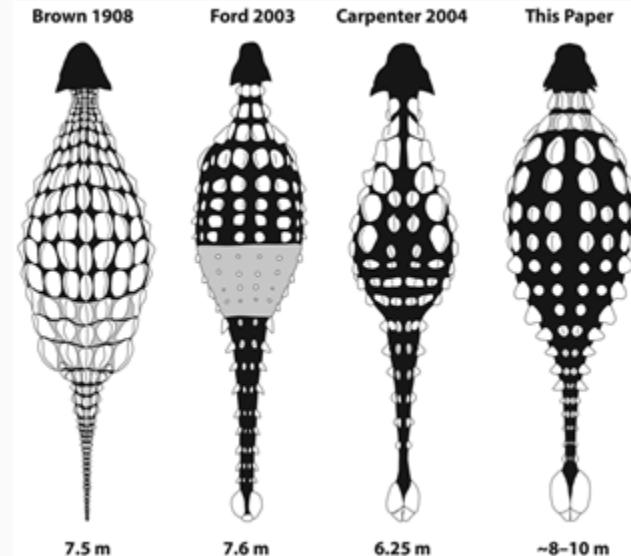
- How would you access the 1000th record?
- How would you access just the third column?

# Record or row-oriented storage: Relational data

- Relational data can be logically **grouped by rows**
  - Each record (tuple) represents a data point
  - Example: CSV files
- This is good if you want to process an entire record at a time
- Also good for appending data
- Easy to read for humans

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

[Arbour & Mallon  
\(2017\)](#)



# Querying row stores

```
SELECT * FROM Dinosaur WHERE Abundant = True
```



```
for row in Dinosaur:  
    if row.Abandant = True:  
        emit row
```

- Each row is loaded from storage (disk)
- Attributes are inspected
- Rows that pass are sent down-stream

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

# Indices can help, but...

```
SELECT Species FROM Dinosaur  
WHERE Abundant = True
```



Now with index on  
Dinosaur.Abandant

```
for row in Dinosaur[Abundant = True]:  
    emit row.Species
```

- An index can help locate rows
- But it still involves pulling an entire row, even if we only want one column
- Loading data from **disk is slow!**

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

# Column-oriented storage

- Each column is stored on its own
- Values in a column have **constant type**
  - Disk access patterns become more regular
  - Enables compression

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False



<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

# Example

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False

## Row-oriented

id, Species, Era, Diet, Abundant

1, T. Rex, Cretaceous, Carnivore, True

2, Stegosaurus, Jurassic, Herbivore, True

3, Ankylosaurus, Cretaceous, Herbivore, True

## Column-oriented

id: [1, 2, 3]

Species: ["T.Rex", "Stegosaurus", "Ankylosaurus"]

Era: ["Cretaceous", "Jurassic", "Cretaceous"]

Diet: ["Carnivore", "Herbivore", "Herbivore"]

Abundant: [True, True, False]

# But there is more: Speed/time is not everything!

- Storage/space matters too (although it often translates to speed gains)
- Mixed types are the enemy of compressibility
- Once the data are arranged in columnar fashion, they also have the same type
- This aids in compression, which saves space and if compressed data is sent/retrieved, time as well.

# Compression

- **Records have heterogeneous types**
- A **single column** only has **one type**
- **Low entropy** in a column ⇒ **compression**
  - Compressed columns take less space
  - Compressed columns are **cheaper to load**
  - Sometimes we can **compute directly on compressed columns!**
- **But what kind of compression scheme should we use?**

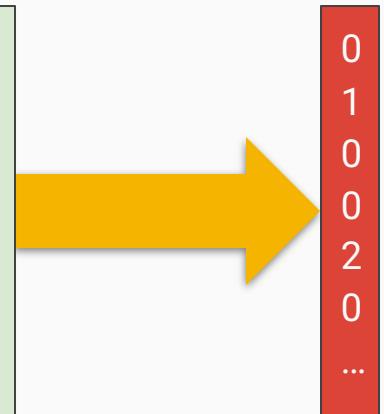
<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>	<b>Mass</b>
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

# Dictionary encoding

<b>id</b>	<b>Species</b>	<b>Era</b>	<b>Diet</b>	<b>Abundant</b>	<b>Mass</b>
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- Useful when you have an attribute which takes **few distinct values**
- Replace **string values** by **string identifiers**
- Column now has **uniform data width**  
    ⇒ better cache locality!
- String matching can be done on the **dictionary**, not each row

Cretaceous  
Jurassic  
Cretaceous  
Cretaceous  
Triassic  
Cretaceous  
...



Cretaceous ↔ 0  
Jurassic ↔ 1  
Triassic ↔ 2

# Bit-packing

- Integers usually consume 4, or 8 bytes (32 or 64 bits)
- Bit-packing squeezes small integers together

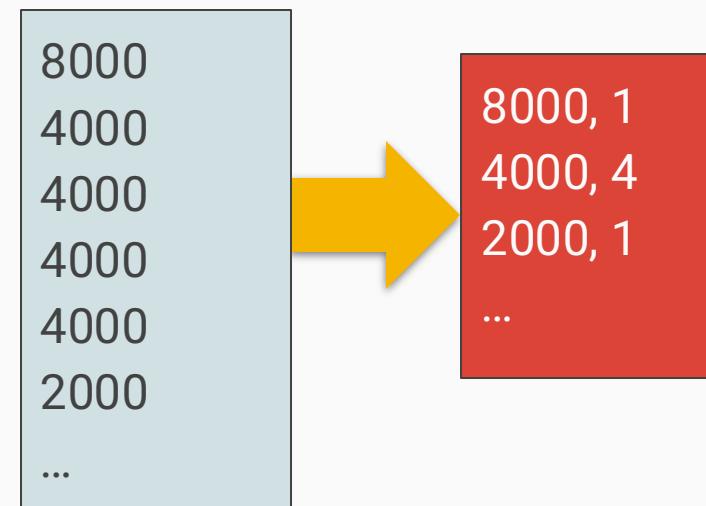
Values	0	1	0	2	1	1
8-bit (binary)	0000 00 <u>00</u>	0000 00 <u>01</u>	0000 00 <u>00</u>	0000 00 <u>10</u>	0000 00 <u>01</u>	0000 00 <u>01</u>
Compressed	  00 <u>01</u> 00 <u>10</u>	 01 <u>01</u> ....				

- Matching and comparing can be done on **compressed values**

# Run-length encoding

id	Species	Era	Diet	Abundant	Mass
1	T. Rex	Cretaceous	Carnivore	True	8000
2	Stegosaurus	Jurassic	Herbivore	True	4000
3	Ankylosaurus	Cretaceous	Herbivore	False	4000

- Useful when you have long runs of a constant value (e.g. DNA, methylation)
- Convert **sequence of values** to tuples (**value, # repetitions**)
- Sums, averages, counts, etc. can all be done on **compressed values**



# Compression schemes abound...

- Frame of reference coding
  - 1004, 1005, 1006 ⇒ **1000** | 4, 5, 6
- Delta coding
  - 1004, 1005, 1006 ⇒ **1004** | +0, +1, +1
- Many others, e.g. “Huffman coding” (relating code word length to inverse of frequency)

Compression schemes can be **combined!**

Delta + bit packing

Dictionary + Run-length encoding

Main trade-off is **space efficiency** vs.  
**complexity of querying/processing.**

# So when should you use which compression scheme?

- The fact that many viable schemes exist implies that it depends.
- And it does – on the use case, the data, the goals, etc.
- There are often even multiple reasonable answers (and many non-reasonable choices)
- For instance, imagine that you work as a data scientist for Netflix and need to compress a movie to stream it efficiently.
- What is the most suitable compression scheme? Hint: A motion picture consists of many successive frames.

# Column oriented storage take-aways

## Pros:

- Can be much faster when you only want a **subset of attributes**
- Higher **storage efficiency** and **throughput**
- Collecting **data of the same type** enables compression and better access patterns


## Cons:

- Reconstructing full tuples can be slow
  - Not great for **record-oriented** jobs
- Writes / deletion can be slow
- **Handling non-tabular data is tricky**


But what if our  
data isn't tabular?

(Spoiler: It often is not!)

# Enter Dremel and Parquet



# Now

- Background on column stores
- **Dremel and Parquet**

**TLDR:** parallelism isn't everything.

Data structures are still important!

# Dremel

[Melnik et al., 2010]

- Low-latency query system for read-only, **structured data**
- Developed at Google ~2006-2010
- Lots of cool ideas in the paper, but we'll focus on the **data format**
- Core ideas were quickly adopted and re-implemented in **Parquet** (2013)
  - Parquet is the default storage format for Spark

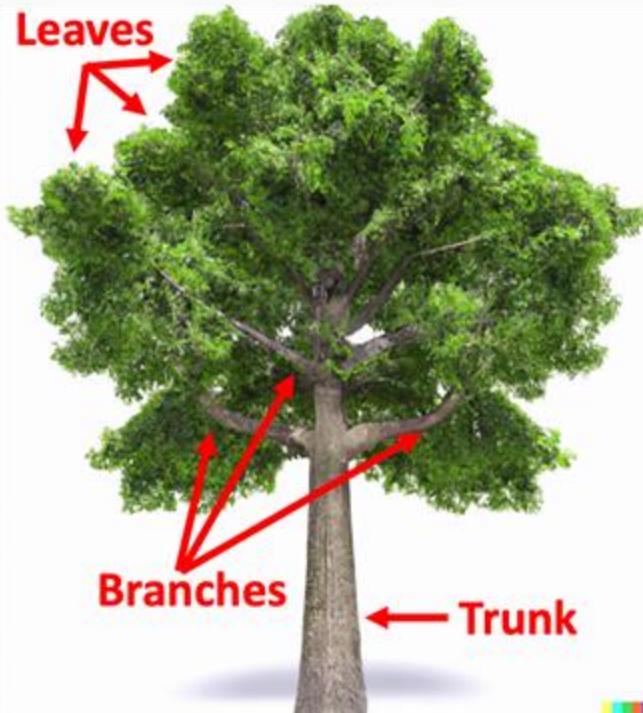
# Nested and structured data

- Not everything fits nicely in relations
- Variable-length/depth can be difficult
- Record-oriented storage is more natural here

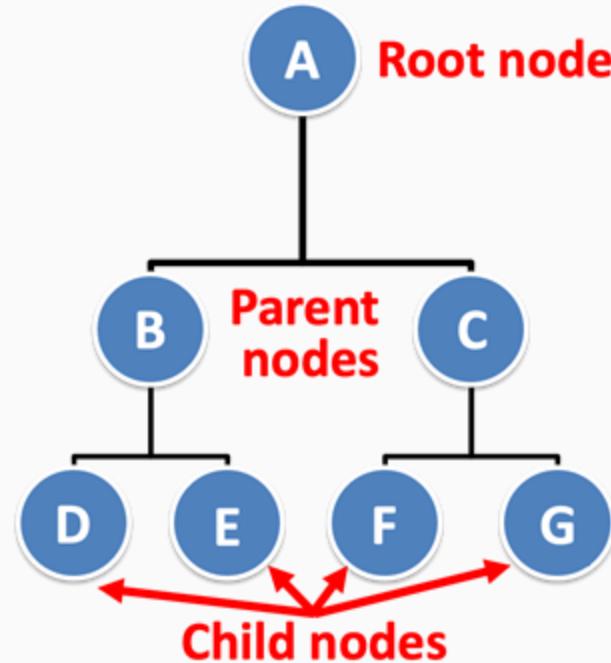
**How can we get all the benefits of column stores but for structured data?**

# A typical hierarchical computer science data structure: Trees

*An actual tree*

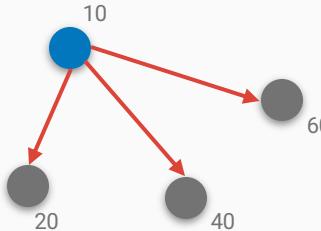


*What a computer scientist thinks a tree looks like*



# Example: web documents

- DocID [required]
- Links [optional]
  - Backward [0 or more]
  - Forward [0 or more]
- Name [1 or more]
  - Language [1 or more]
    - Code [required]
    - Country [optional]
  - URL [optional]



DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

# Example: web documents

- DocID [required]
- Links [optional]
  - Backward [0 or more]
  - Forward [0 or more]
- Name [1 or more]
  - Language [1 or more]
    - Code [required]
    - Country [optional]
  - URL [optional]

Field names are *paths*, e.g.:

Node.DocID  
Node.Links.Forward  
Node.Name.Language.Code



The level of a field in the hierarchy can be gleaned from the dot number (indexed from 0)

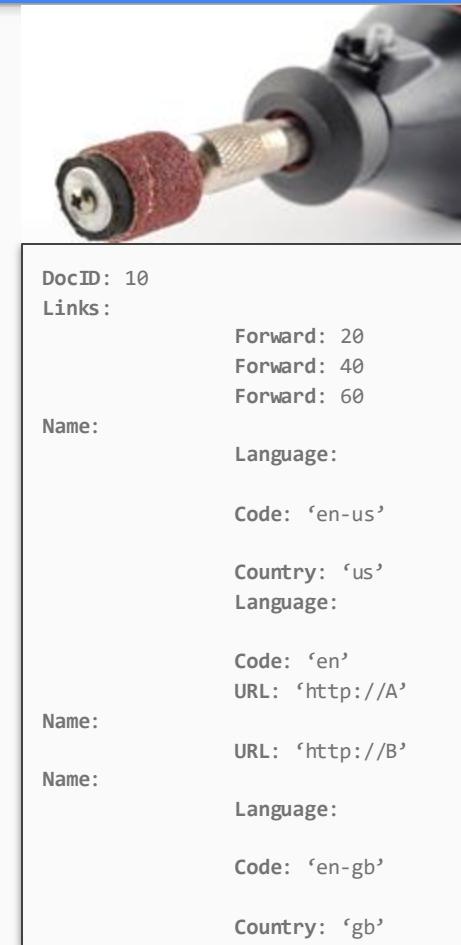
DocID:	10
Links:	
Forward:	20
Forward:	40
Forward:	60
Name:	
Language:	
Code:	'en-us'
Country:	'us'
Language:	
Code:	'en'
URL:	'http://A'
Name:	
URL:	'http://B'
Name:	
Language:	
Code:	'en-gb'
Country:	'gb'

## What specs would we like to see in a system that “flattens” hierarchical records?

- Lossless representation of the hierarchical record in columnar format
- It needs to be possible to recreate the hierarchical records from the columnar format
- Key challenge: Being able to parse records that have an arbitrary structure **\*unambiguously\***.
- Need to be able to keep track of the record structure, e.g. if a value appears in a table twice, we need to be able to denote whether it is in the same record twice (“repeats”) or whether the value came from two different records.
- To make this efficient, it needs to be able to handle sparse datasets
- We need to be able to represent missing fields efficiently (e.g. with NULL values)

# Implementing record flattening with Dremel

- **Key idea:** keeping track of repetitions of fields within a record
- **Repetition level ( $r$ ):** at which level of the hierarchy did the repetition happen?
- **Definition level ( $d$ ):** how many optional fields in the path to a given field are present (up to deepest nested optional field)?
- **Required fields  $\Rightarrow$  d-level is determined by how many optional fields are logically traversed to reach the required field**
- **Optional fields  $\Rightarrow$  Same r-level as parent, d-level as in the definition above, but also increments (only!) IF present**
- **Repeated fields  $\Rightarrow$  r-levels increment from parent, d-levels increment IF they are optional**



# There is much potential for confusion - addressing common misconceptions:

- Note: "**Repetition level**" does *\*not\** count how many times a field repeats, instead, it simply represents at which *\*level\** of the hierarchy the repetition happened.
- Note: The Dremel system needs to **both** represent the values of the document as well as the (tree) structure of the document in tabular format, thus numbers in the table will have *\*different\** meanings.
- Note: It is important to **distinguish** between required and optional fields, as **different** rules to determine r- and d-levels apply.
- Note: Optional present vs. optional absent yields different definition levels to distinguish between possibly present (but actually absent) information and information that is missing – and can't be added.
- Note: We will represent the hierarchy of the document with paths, for instance Node.Name.Language.Country. The number of dots indicates the level of the field in the hierarchy, indexed from 0 (0,1,2)
- Note: The d-level helps to disambiguate between NULL values and the absence of values. For instance, a middle name field. Might have a middle name, but wasn't mentioned, vs. does not have a middle name.



DocID:	10
Links :	Forward: 20 Forward: 40 Forward: 60
Name:	Language:
	Code: 'en-us'
	Country: 'us'
	Language:
Name:	Code: 'en'
	URL: 'http://A'
Name:	URL: 'http://B'
	Language:
	Code: 'en-gb'
	Country: 'gb'

# Flattening example

**Node.DocID**

value	r	d
-------	---	---

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
-------	---	---

**Node.Links.Backward**

value	r	d
-------	---	---

**Node.Name.Language.Code**

value	r	d
-------	---	---

**Node.Name.Language.Country**

value	r	d
-------	---	---

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d

**Node.Links.Forward**

value	r	d

**Node.Links.Backward**

value	r	d

**Node.Name.Language.Code**

value	r	d

**Node.Name.Language.Country**

value	r	d

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID is required

r=0, d=0

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

Node.DocID

value	r	d
10	0	0

Node.Name.URL

value	r	d

Node.Links.Forward

value	r	d
20	0	2

Node.Links.Backward

value	r	d

Node.Name.Language.Code

value	r	d

Node.Name.Language.Country

value	r	d

DocID: 10

Links:



Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

Links is **optional** (but present)

First occurrence  $\Rightarrow r=0$

Links.Forward is a **repeated** field  
Forward  $\Rightarrow d=2$

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2

**Node.Links.Backward**

value	r	d
-------	---	---

**Node.Name.Language.Code**

value	r	d
-------	---	---

**Node.Name.Language.Country**

value	r	d
-------	---	---

DocID: 10

Links:



Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

...Forward  $\Rightarrow$  d=2

Repetition in level r=1

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
-------	---	---

**Node.Name.Language.Code**

value	r	d
-------	---	---

**Node.Name.Language.Country**

value	r	d
-------	---	---

DocID: 10

Links:

Forward: 20

Forward: 40

**Forward: 60**

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

...Forward  $\Rightarrow d=2$

Repetition in level r=1

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
-------	---	---

**Node.Name.Language.Country**

value	r	d
-------	---	---

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name :

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name :

URL: 'http://B'

Name :

Language:

Code: 'en-gb'

Country: 'gb'

Links.Backward is **repeated** (but absent)

r=0, d=1

No value in this record, so fill a NULL

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name :

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2

**Node.Name.Language.Country**

value	r	d
-------	---	---

Name.Language.Code required

First occurrence (r=0)  
Full definition path (d=2)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2

**Node.Name.Language.Country**

value	r	d
us	0	3

...Country is optional  $\Rightarrow d=3$

First occurrence ( $r=0$ )  
Full definition path ( $d=3$ )

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'



Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2

**Node.Name.Language.Country**

value	r	d
us	0	3

...Code is required

Repetition at r=2  
(Name.Language)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

**Code: 'en'**

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'



# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
-------	---	---

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2

...Language.Country optional

Repeated at Language level  
r=2, d=2

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name :

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name :

URL: 'http://B'

Name :

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name :

URL: 'http://C'



# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2

Node.Name.URL is optional  
 $\Rightarrow d=2$

No repetitions: r=0

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'



Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2

**Node.Links.Forward**

value	r	d
20	0	2
40		
60		

**Node.Links.Backward**

value	r	d
NULL	0	1

Node.Name  $\Rightarrow$  d=1

But no Language...

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2
NULL	1	1

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2
NU		

Node.Name.URL  $\Rightarrow$  d=2

Repetition at r=1 (Node.Name)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2
NU		

...Language.Code  $\Rightarrow$  d=2

Repetition at r=1 (Node.Name)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2

**Node.Links.Forward**

value	r	d
20	0	2
40		
60		

**Node.Links.Backward**

value	r	d
NULL	0	1

...Language.Country  $\Rightarrow d=3$

Repetition at r=1 (Node.Name)

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

**Country: 'gb'**



DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2

Node.Name  $\Rightarrow$  d=1

No URL data

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0
20	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2

Node.DocID  $\Rightarrow$  d=0

Required field, new document  
(r=0)

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0
20	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2

**Node.Links.Backward**

value	r	d
NULL	0	1
10	0	2

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2

Node.Links.Backward  $\Rightarrow d=2$

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10



Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Flattening example

**Node.DocID**

value	r	d
10	0	0
20	0	0

**Node.Name.URL**

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

**Node.Links.Forward**

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

**Node.Links.Backward**

value	r	d
NULL	0	1
10	0	2
30	1	2

**Node.Name.Language.Code**

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

**Node.Name.Language.Country**

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

... and all  
the rest

DocID: 10

Links:

Forward: 20

Forward: 40

Forward: 60

Name:

Language:

Code: 'en-us'

Country: 'us'

Language:

Code: 'en'

URL: 'http://A'

Name:

URL: 'http://B'

Name:

Language:

Code: 'en-gb'

Country: 'gb'

DocID: 20

Links:

Backward: 10

Backward: 30

Forward: 80

Name:

URL: 'http://C'

# Be careful!

ChatGPT o3-mini-high ▾

- **Definition Levels:** Capture whether a field is present or missing at a particular nesting level. They help record the "depth" at which a field exists in the hierarchy.
- **Repetition Levels:** Track how many times a repeated field (e.g., an array) occurs. They help maintain the relationship between repeated elements.

Are you sure that this is what the repetition level represents? How MANY times a repeated field occurs? I thought it is at which level of the hierarchy the repetition happened.

Reasoned for 9 seconds >

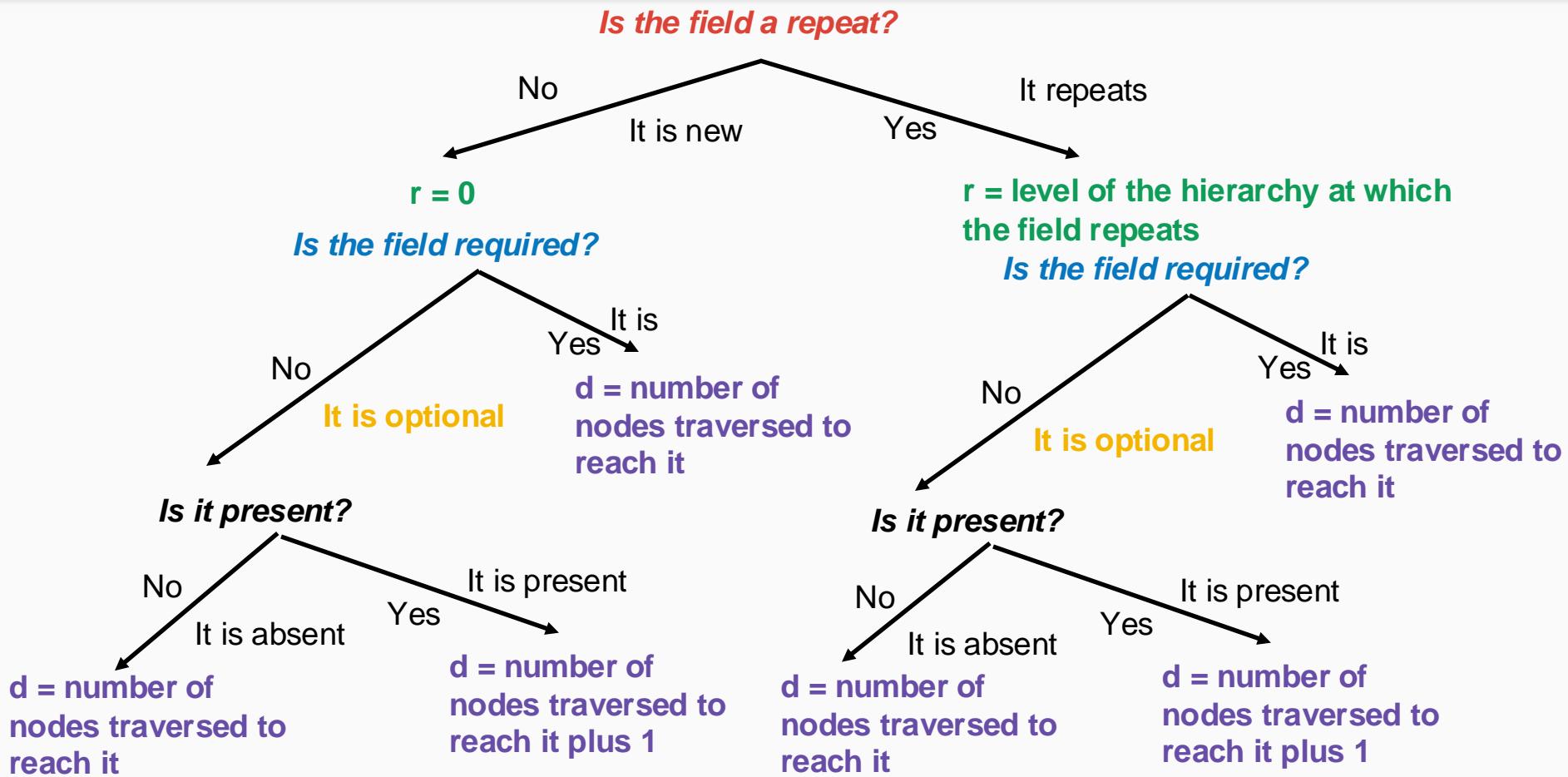
You're right—the repetition level in Dremel's encoding isn't repeated. Instead, it encodes information about the level it is at. In other words, it tells you at what depth of the nested representation a field occurs—which in turn helps indicate whether a value is a continuation of a previous value or a new one.

So - just to be clear - your initial response on repetition levels was a hallucination. I don't blame you - it is what would make sense on the surface level (but is not true). I'm asking because I want to show my students to beware of hallucinations

Reasoned for 9 seconds >

Yes, that's correct. My initial explanation was an oversimplification and not entirely accurate. In Dremel's encoding, the repetition level indicates the depth in the nested schema where a repeated field occurs—not a literal count of how many times the field is repeated. This level information helps determine whether a particular value is a continuation of a repeated group or starts a new one.

CDS: Flowchart as to how Dremel works: For \*each\* new line of the record:



# After flattening...

- Repetition and definition columns are highly compressible
- Value fields are now columnar / of the same type
  - May also be compressed
- Columns are broken into **blocks** and compressed independently
  - This alleviates some decoding complexity and improves parallelism

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

From Dremel  
to  
Parquet

# Parquet:

## A columnar storage format

- Developed at Twitter and Cloudera, v1.0 in 2013
- Now an Apache project, and the default/recommended storage for Spark. Has become a de facto industry standard for storing large amounts of data.
- Based on Dremel flattening, but without the analysis engine / query mechanism
- The name comes from the shape of the data:
  - blocks of column fragments
  - reminiscent of a popular floor tiling style



# An example from my work: Music Data Science

Row based storage (e.g. csv): 8 rows, 5 columns = 40 cells to scan

	Artist	Title	Genre	Release Year	Duration in sec
Row 1	Beethoven	Für Elise	Classical	1867	177 ➔
Row 2	Martina Mc Bride	Independence Day	Country	1994	204 ➔
Row 3	David Guetta	Titanium	Electronic	2011	232 ➔
Row 4	Michael Jackson	Billie Jean	Pop	1983	294 ➔
Row 5	Mozart	Requiem in D Minor	Classical	1791	416 ➔
Row 6	Taylor Swift	Shake it off	Pop	2014	217 ➔
Row 7	Sara Evans	A little bit stronger	Country	2010	258 ➔
Row 8	Darude	Sandstorm	Electronic	1999	446 ➔

# Query: Average duration as a function of genre

Column based storage: 8 rows, 2 columns = 16 cells to scan

Column 1	Column 2	Column 3	Column 4	Column 5
Artist	Title	Genre	Release Year	Duration in sec
Beethoven	Für Elise	Classical	1867	177
Martina Mc Bride	Independence Day	Country	1994	204
David Guetta	Titanium	Electronic	2011	232
Michael Jackson	Billie Jean	Pop	1983	294
Mozart	Requiem in D Minor	Classical	1791	416
Taylor Swift	Shake it off	Pop	2014	217
Sara Evans	A little bit stronger	Country	2010	258
Darude	Sandstorm	Electronic	1999	446

# Parquet has another level of organization: Row groups

Query: Select genre, duration where genre = country, pop

Parquet: 4 rows, 2 columns = 8 cells to scan

	Column 1 Artist	Column 2 Title	Column 3 Genre	Column 4 Release Year	Column 5 Duration in sec
Row group 1	Beethoven	Für Elise	Classical	1867	177
	Mozart	Requiem in D Minor	Classical	1791	416
Row group 2	Martina McBride	Independence Day	Country	1994	204
	Sara Evans	A little bit stronger	Country	2010	258
Row group 3	David Guetta	Titanium	Electronic	2011	232
	Darude	Sandstorm	Electronic	1999	446
Row group 4	Michael Jackson	Billie Jean	Pop	1983	294
	Taylor Swift	Shake it off	Pop	2014	217

# There is more: Parquet also allows for data compression

In particular: Dictionary encoding

0 Classical

1 Country

2 Electronic

3 Pop

**Column 1**  
**Artist**

**Column 2**  
**Title**

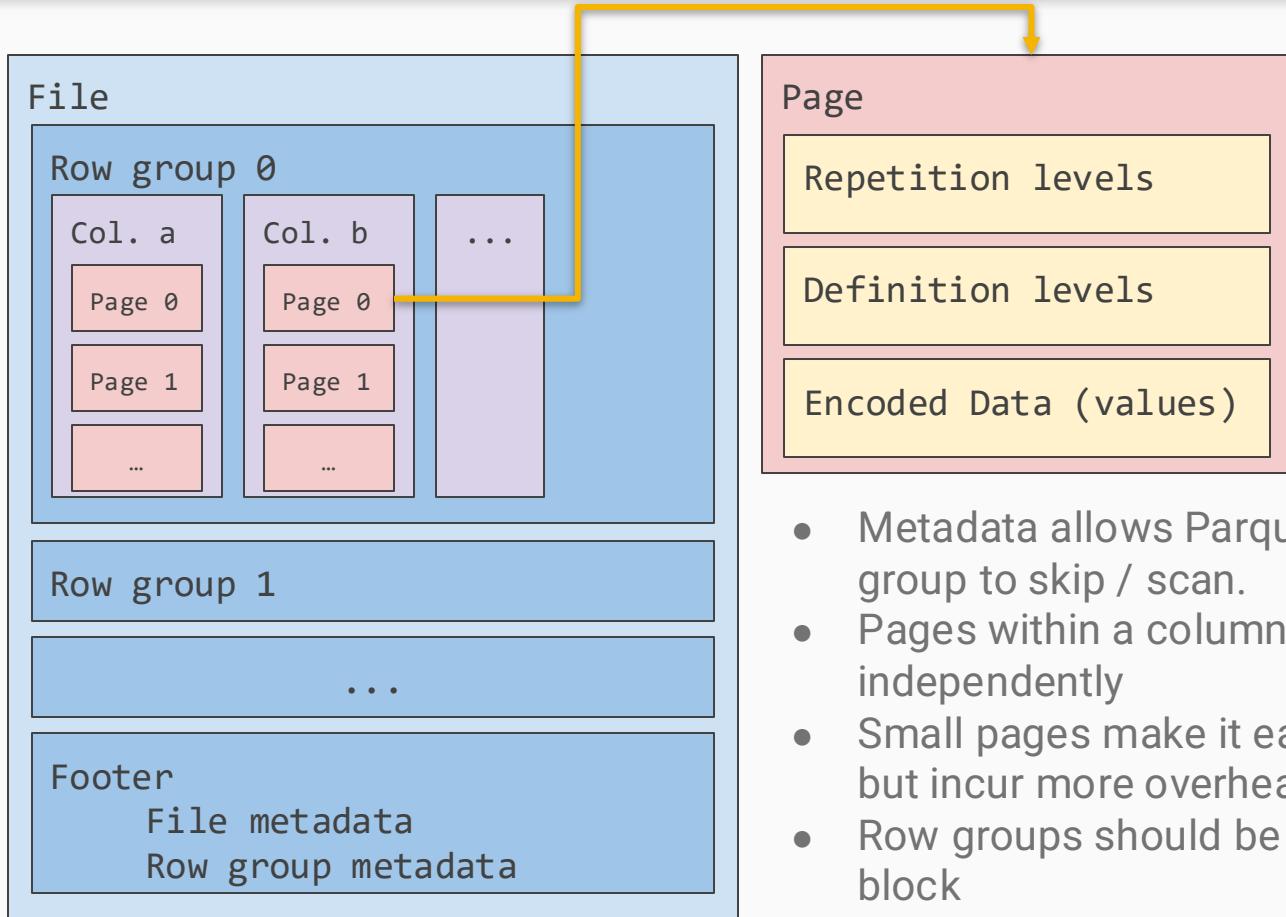
**Column 3**  
**Genre**

**Column 4**  
**Release Year**

**Column 5**  
**Duration in sec**

Row group 1	Beethoven	Für Elise	Classical	1867	177
	Mozart	Requiem in D Minor	Classical	1791	416
Row group 2	Martina Mc Bride	Independence Day	Country	1994	204
	Sara Evans	A little bit stronger	Country	2010	258
Row group 3	David Guetta	Titanium	Electronic	2011	232
	Darude	Sandstorm	Electronic	1999	446
Row group 4	Michael Jackson	Billie Jean	Pop	1983	294
	Taylor Swift	Shake it off	Pop	2014	217

# Parquet format: Storage Principles



- Metadata allows Parquet to determine which row group to skip / scan.
- Pages within a column are compressed independently
- Small pages make it easier to read single records, but incur more overhead
- Row groups should be large, but fit in one HDFS block

# Good and bad things about Parquet

- Cross-platform, cross-language support
  - Java, C++, Python, Scala, ...
- Allows **partial decoding** (only decode necessary columns)
- Integrates nicely with Spark and HDFS
  - Preserves RDD / DataFrame schema directly
  - HDFS block-aware layout
  - Partition discovery / exposes control over partitions by column
- Cost: Complex scheme and not human-readable (binary file format)

# You are probably already using column-oriented storage

- Most DataFrame implementations are columnar (pandas, R)
  - This is the most reasonable way to handle large mixed-type data!

# Wrap-up

- Data representation matters too.
- Column stores can improve speed for attribute (rather than record)-oriented analyses
- Dremel turns structured or variable-length data into columnar representations
- Parquet provides an open source reimplementation of the Dremel format

# Next up...

- Dask (last software/framework)
- Then on to other things:
  - Data structures
  - Algorithms
  - Applications

**Q & R**



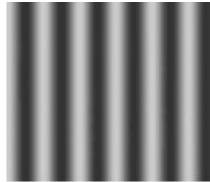
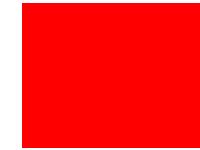
Smallest font



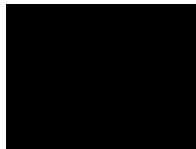
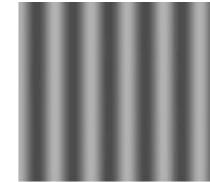
Please turn off and put  
away your cell phone



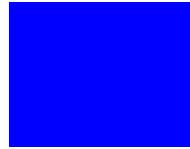
# Calibration slide



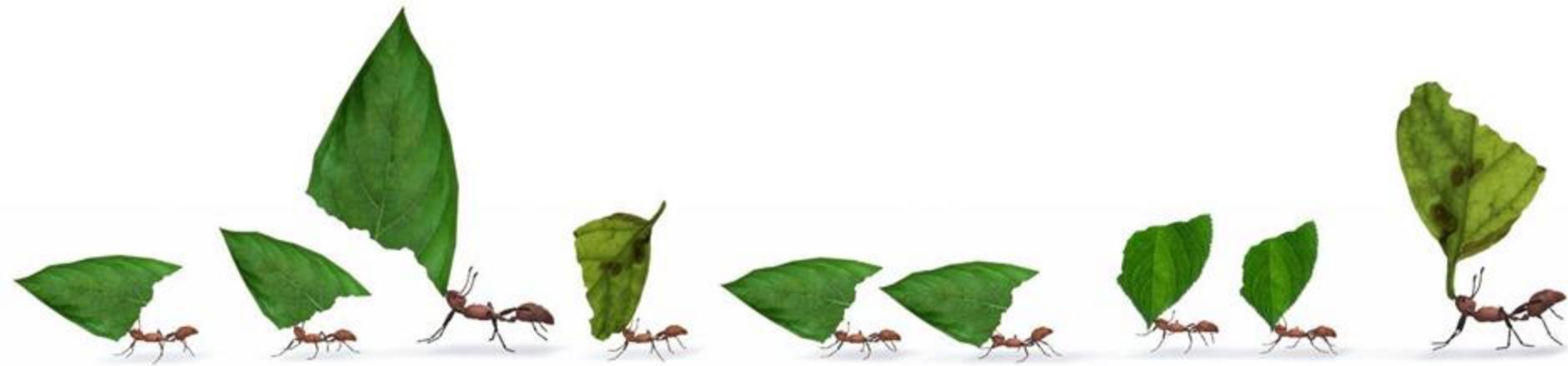
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



# Week 08: Dask

DS-GA 1004: Big Data

# Announcements

- HW4 (Spark) will be released within the next couple of days
- Outstanding grades will also be released this week

# The story so far...

How do we store and process large collections of data?

- **Collections of files**
  - Unstructured
  - Lots of custom coding
  - Very flexible
  - Parallelism? *You're on your own...*
- **Relational databases**
  - Restricted / structured (tabular) data
  - Standard interface (SQL)
  - Somewhat flexible
  - Parallelism? *It's complicated...*
- **Map-Reduce + HDFS**
  - Data is less structured than RDBMS
  - Restricted coding interface (map/reduce)
  - Very parallel!
- **Spark**
  - Structured data like RDBMS
  - Distributed storage (HDFS)
  - Standard-ish interface (SQL or Spark object API)
  - Very parallel!

# Spark is very useful for many things!

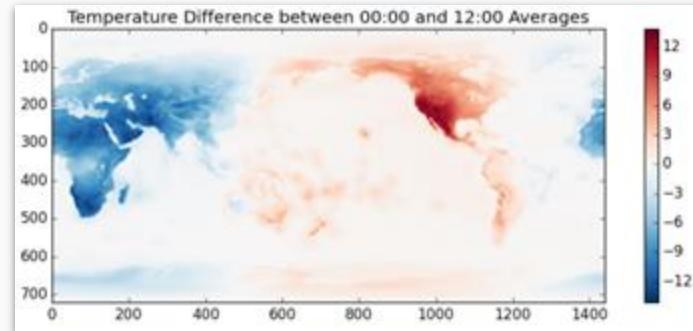
- Spark integrates nicely with Java-based tools (**Hadoop ecosystem**)
  - HDFS, Parquet, YARN scheduler, etc...
- Spark is great for DataFrames and SQL-like processing
  - *Graphs also, though we haven't gotten to that yet – we'll see this later in the course*
- >10 years old now, **implementation is mature and stable**
- After RDBMS/SQL, **probably the most widely used software** we cover in this course

# But Spark is not useful for everything...

- Can you think of a use case where Spark would be unsuitable or a computation that would be difficult to do with Spark?

# Some things are still difficult with Spark...

- Data that doesn't naturally fit the RDD/DataFrame model:
- Integration with scientific Python stack
- (Modern) Machine learning tools:
  - sklearn, pytorch, etc...
- Scaling **down** vs. scaling **out**:
  - What if the data fits into laptop storage, but not RAM?
  - Do you really need a cluster?



**Macintosh HD** Capacity: 4 TB

Modified: February 2, 2024, 12:19 PM

Memory

64 GB 2667 MHz DDR4

# Dask: An open source parallel computing library

[Rocklin, 2015]

- Python-based distributed computation
- Many common design principles with **Spark**
  - Computation graphs (**lineage graphs**)
  - Delayed computation (**RDD/transformations**)
  - Collections-based interfaces (**DataFrames**)
- Some key differences from Spark:
  - **Prioritizes array-based (numpy-like) computation**
  - **Designed to support single-machine, out-of-core use**

**“Out-of-core computation”:**

Processing data that exceeds the size of main memory of the computer

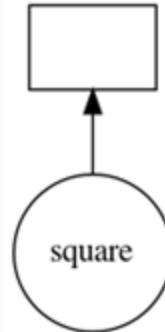
# Delayed computation and task graphs in Dask

- Dask builds complex computations by composing deferred computations into a **task graph**. Like in Spark, nothing happens until you take an **action**.

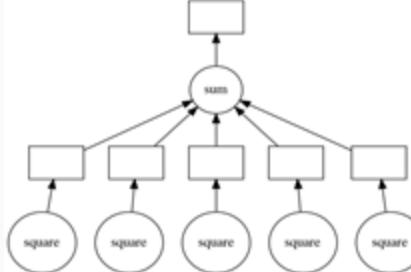
```
import dask

def square(x):
    return x**2

f = dask.delayed(square)
y = f(5)
y.visualize() # draw computation graph
```



```
g = dask.delayed(sum)
z = g([f(x) for x in range(5)])
z.visualize()
z.compute()
```



# There are several types of collections in Dask

- **Bags**
  - Distributed collections of arbitrarily structured data
  - **Most similar to:** `RDD`
- **DataFrames**
  - Distributed collection of structured, tabular data
  - **Most similar to:** `Spark DataFrame` (but built on pandas instead of RDDs)
- **Arrays**
  - Distributed n-dimensional arrays
  - **Most similar:** to `numpy.ndarray` (but distributed!)

# Collections interfaces: Bags

- Dask **bags** are loosely analogous to Spark RDDs
  - But you can think of it more like a (parallel) python list
- Unordered collection of generic Python objects
  - Partitions into subsets (sub-bags) to parallelize
- Implements some basic operations
  - map, filter, join, sum, etc.
- A good choice for initial processing and handling of structured objects
  - If your data is tabular or array-based, there are better choices than bags

```
import dask.bag as db

b = db.from_sequence(range(5))

c = b.map(square)

c.compute() # [0, 1, 4, 9, 16]

c.sum().compute() # 30
```

# Dask Bags vs. Spark RDDs: Similarities and differences

- Both allow to partition a collection across multiple machines / cores
- Both are immutable
- RDDs have **types** (e.g. `RDD[Integer]`)
- Bags are **untyped**, contents can be mixed

# Tips for working with bags

- A common workflow is:  
Raw data → Bags → DataFrame → (more sophisticated analysis)
- The earlier you can reduce the size of your data, the better!
  - Less data moving through the system is a good thing!
- Recommended: **maps** and **filters** on bags over DataFrame operations when simplifying data

# Tips for working with bags

- Common workflow is:  
Raw data → Bags → DataFrame → (more sophisticated analysis)
- The earlier you can reduce the size of your data, the better!
  - Less data moving through the system is a good thing!
- Prefer **maps** and **filters** on bags over DataFrame manipulations when simplifying data
- **HOWEVER**, bag operations are generally slower than DataFrame operations
  - For the same reason that pandas/numpy is faster than vanilla python code
- Because bags have so few restrictions, Dask can't assume much.
  - You'll have to think carefully about optimizing your code.

# Another practical tip: Bag folding vs grouping

```
import dask.bag as db

b = db.from_sequence(range(10))
iseven = lambda x: x % 2 == 0
add = lambda x, y: x + y

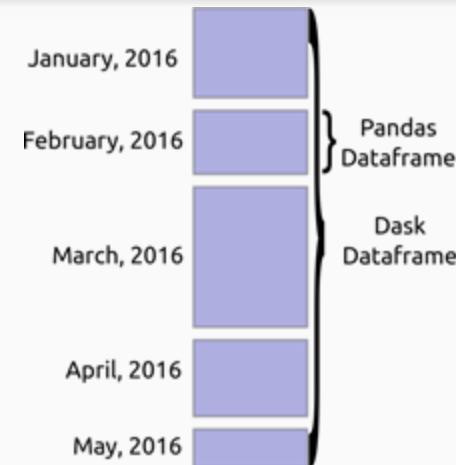
dict(b.foldby(iseven, add))
```

- Try to **avoid using groupBy** on bags
  - This requires much inter-worker communication, and is slow!
  - This is the “wide dependency” problem from Spark again
- Use **fold/foldBy** if possible
  - Similar benefits to a **combiner** in map-reduce
  - Perform local aggregation (within partition) first to reduce shuffling
  - Same restrictions apply (associative, commutative)
- You supply a **key function** and a **binary operation**

# Collections interfaces: **DataFrames**

- Just like you'd expect, similar to Spark DataFrames
  - Uses Pandas internally, interface is basically the same
- Parallelism (partitioning) is over subsets of **rows**.  
Each partition is a Pandas dataframe
- Good choice for data that can naturally split into multiple CSV files (or Parquet partitions)

```
import dask.dataframe as dd  
  
df = dd.read_csv('*.csv')  
df.mean().compute()
```



# Dask DataFrames and partition management

- **Managing your partitions is important!**
  - Think about how your data changes through the computation graph
- If you're **filtering out records**, you may end up with many (near) **empty partitions**
- Try to keep your partitions **full and balanced**
- Expect to work harder than in Spark for good performance

```
df = dd.read_csv('s3://bucket/path/to/*.csv')

df = df[df.name == 'Alice'] # only 1/100th of the data

df = df.repartition(npartitions=df.npartitions // 100)

df = df.persist() # if on a distributed system
```

# Aside

Here's a real-life example how one could use Dask Dataframes

# Example: machine listening evaluation

- Say you have 10 models for automatic audio segmentation that you want to compare
- You have a dataset of 2000 audio recordings  
⇒ **20,000 model outputs** to evaluate
- (Pre-computed) **model outputs** are, for each recording, a sequence of labeled time intervals:

# start_time,	end_time,	label
0,	5,	silence
5,	23.2,	intro
23.2,	44.9,	verse
...		
- Model evaluator compares **reference annotation** to **estimate annotation** for one track, and produces a dictionary of scores along different metrics.
- What you want: a DataFrame containing:  
*model id, recording id, [scores for each metric]*

## Example: machine listening evaluation

- Solution:
  - Store model all outputs on disk as “`{model_id}/{recording_id}.txt`”  
`files = glob('*/*.*txt')`
  - Create a delayed function to map filename to scores (calls the evaluator)  
`my_evaluator(file_name):`  
    `# load estimate and reference data`  
    `# call the evaluator`  
    `return scores_dict`  
`evaluator = delayed(my_evaluator)`
  - Create a bag from the delayed function  
`results = db.from_delayed([evaluator(estimate) for estimate in files])`
  - Convert the bag to a dataframe and save  
`results_df = results.to_dataframe(...) # not pictured: schema definition`  
`results_df.to_parquet('output.parquet')`

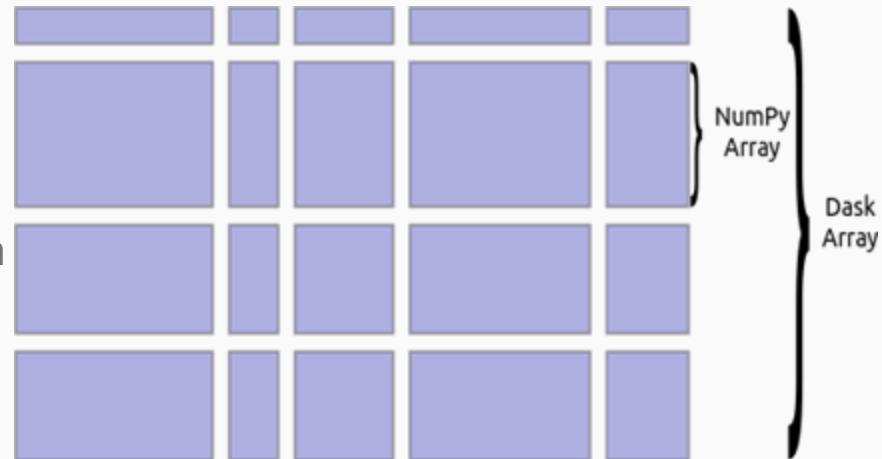
# What did this save us?

- Everything is done with basic python functions
  - Core computation (evaluator) was not written explicitly for Dask
- The problem itself is “embarrassingly parallel”
  - aka “all map, no reduce”
  - Don’t need to think too hard about partition structure
- Coding this up with Dask is about as simple as it gets
  - Probably could also be done with MrJob, but with a lot more overhead

Back to other  
Dask  
collections  
interfaces...

# Collections interfaces: Arrays

- Dask Arrays work like NumPy arrays
- **Parallelism is not limited to rows**
  - You can define **chunks** along each dimension
- Large arrays are assembled implicitly from many small arrays
- Most\* numpy operations work automatically



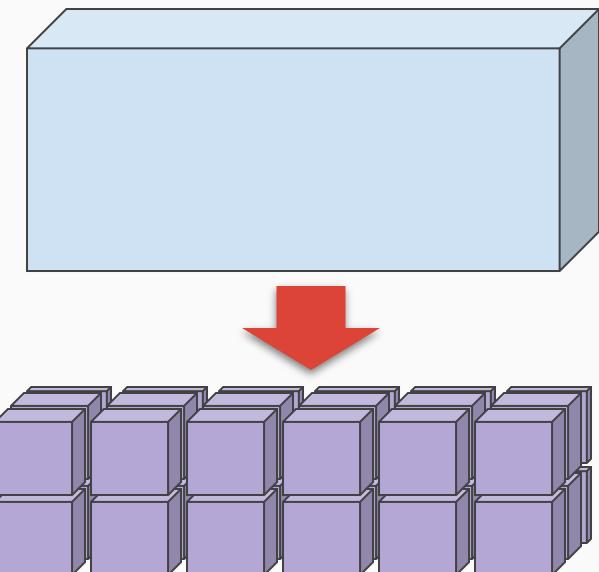
Example image from <https://docs.dask.org/en/latest/array.html>

# Array chunking example

```
import numpy as np
import dask.array as da

# Make some random noise: 2000 x 6000 x 5
x = np.random.randn(2000, 6000, 5)

# Slice it up into chunks
x_parallel = da.from_array(x, chunks=(1000, 1000, 2))
```



# Scaling down: why a single machine?

- **Many “big data” jobs do not really need a cluster!**
  - Data fits on a hard drive, but not in RAM (“core memory”)
  - NumPy (& friends) generally assume fully observed, in-memory data
- In this case, working on small chunks at a time is sufficient
  - Coding this by hand can be tedious / error-prone
- Dask simplifies this, and makes it easy to migrate to a cluster if necessary

# Numpy vs. Dask arrays: Simplicity vs. scalability

- If the problem fits into memory (data is not prohibitively large), numpy outperforms Dask, due to lack of overhead (scheduling, managing threads, etc.)
- If the data exceeds system memory, a dask array might be more suitable. Under the hood, a dask array consists of chunks, each of which is a numpy array.
- Processing of these chunks with Dask can be parallelized, both on a single machine and on a cluster (more on this on the next slide).
- Numpy is not inherently multi-threaded, whereas Dask is (on a multi-core system, Dask can often yield near-linear speedups, if tasks are embarrassingly parallel).

# Schedulers: how does Dask code run?

- On a single machine, you have several options:
  - Parallelizing into multiple **processes**
  - Parallelizing into multiple **threads**
  - Both
- Or you can run on a cluster
  - Execution can look very similar to Spark (eg YARN jobs)
  - Data is transferred automatically / as needed

## Processes

- An instance of a program with a self-contained execution environment with own resources
- Shared data must be sent between processes
- Processes do not block each other

## Threads

- Components of a single python process
- Shared memory between threads
- Certain python operations can block computation for all threads

# Multi-thread vs. multi-process parallelization

- Modern machines often have multiple cores.
- There are many processes and threads running concurrently
- One thread runs on one core, not split between cores
- However, each core can only execute one thread at a time, putting a limit on true parallelization (threads are competing with each other).
- If that is a concern, cloning the process and distributing them over multiple CPU cores can yield true parallelization.

Processor	2.4 GHz 8-Core Intel Core i9
Threads:	5,081
Processes:	852

# Using Dask with HDF5 (not the same as HDFS!)

## HDF5 = Hierarchical data format, version 5

- Basically a file-system within a file
  - (Hierarchical) Directory structures
- In python, use the **h5py** package
- HDF5 supports memory-mapped file access (non-human readable binaries).
- Data is memory-mapped, not loaded

```
import h5py

data = h5py.File('myfile.h5', mode='r')

x = data['/x']
y = data['/y']
z = data['/path/to/z']
```

```
import dask.array as da

x_parallel = da.from_array(x, chunks=(1000, 1000))
```

# Does Dask replace Spark?

- It depends...
  - <https://docs.dask.org/en/latest/spark.html> summarizes use-cases and differences
- Pros for Dask:
  - Do you need to integrate with the SciPy stack? (Matplotlib, sklearn, etc)
  - Do you need to work with dense / multi-dimensional data?
  - Custom algorithms / advanced machine learning? GPUs?
- Pros for Spark:
  - More mature, possibly more stable / safe
  - More “high-level” -- you don’t need to think as much about the compute graph
  - Probably faster / better optimized for DataFrame crunching
  - Better support for large graph data

# HPC: Dataproc and Greene

- So far, we've been using a Hadoop cluster (**Dataproc**)
  - HDFS storage
  - MapReduce + Spark jobs (YARN)
- We also have the **Greene** cluster for less restrictive computation
  - Network-accessible file storage (IBM GPFS, **not** HDFS)
  - Traditionally preferred if you have “embarrassing parallelism”, but Dask can run on it too

# Dask is not the only parallelization option

- There is an increasing number of parallel versions of classic Python libraries.
- For instance: Polars, a parallel version of Pandas.
- Polars is implemented in Rust, to give C-like speed, but exposed to Python.
- In contrast to Pandas (single-core), Polars runs on all CPU cores.
- Polars: Column oriented storage (via Apache Arrow)
- Polars: Lazy evaluation (optimized query plans)
- Polars: Inherent multi-threading
- Pandas dataframes: Mutable. Polars: Immutable
- ...

Next time  
(next week:  
Spring break)

- Starting applications
- Similarity/search



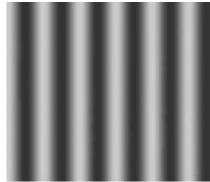
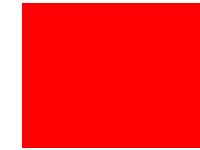
Smallest font



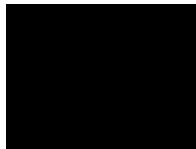
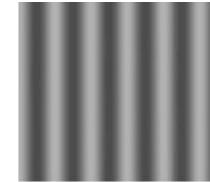
Please turn off and put  
away your cell phone



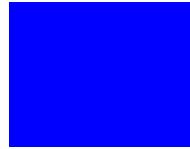
# Calibration slide



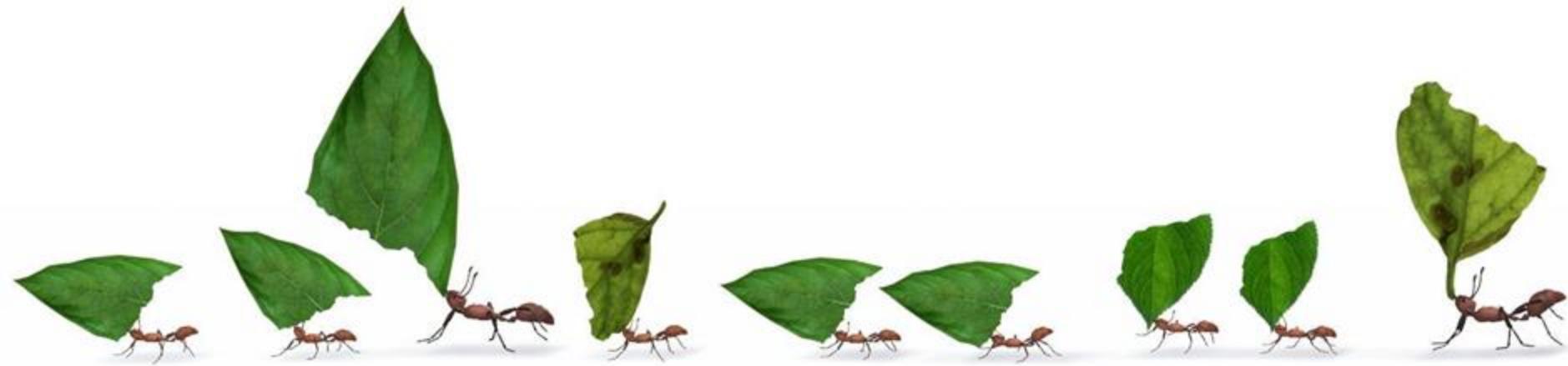
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data



# Week 09: Similarity based search

DS-GA 1004: Big Data

# We are at a pivotal point in this class

- We spent 2/3 of the time in this class on various frameworks to distribute work efficiently, if there is too much work to do for a single worker.
- This is sound (both in terms of distributed data storage and suitably distributed computing) in terms of scaling, reliability, maintainability.
- But there is another approach (which we will spend the last 1/3 of this class on): Working \*smart\*(er).
- However, what “smart” means is often application specific (e.g. search, recommendation, etc.)

If you have a big problem: Looking for a small needle in a large haystack

Is it your only option to hire more workers and/or  
distribute the large stack into smaller piles?



# You could also work smarter:



# Big Data = Big opportunity

- Big Data = Big problem
- Scale brings its own inherent challenges
- Potential solutions:
- Divide the problem into smaller problems, then solve them in parallel.
- Use a more suitable data structure / format
- Work smarter, not harder

# Finding items in a large collection

- **Search and recommendation** rely on similarity calculation
- User provides a “**query**”, e.g.:
  - Search string
  - Example document
  - Latent representation
- System returns a list of matching “**documents**” from the database

# Examples abound

- **Text search:** search string  $\Leftrightarrow$  the web (documents)
- **Recommender systems:** user representation  $\Leftrightarrow$  item representation
- **Reverse image search:** photo  $\Leftrightarrow$  library
- **Copyright detection:** uploaded video  $\Leftrightarrow$  all of youtube
- **Plagiarism detection:** uploaded document  $\Leftrightarrow$  all documents

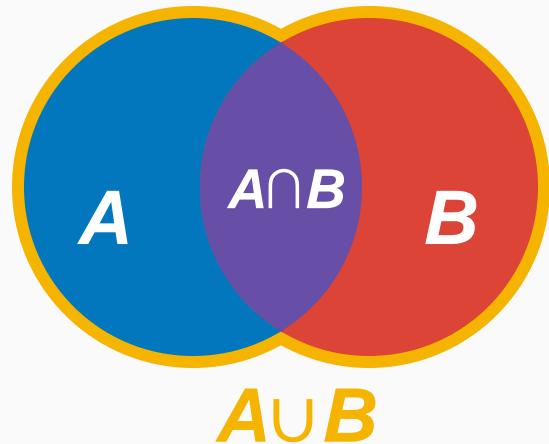
# Basic approach: Conceptually very straightforward

- Given a query  $q$
- For each document  $d$  in collection
  - Compute  $\text{similarity}(q, d)$
- Order collection by decreasing **similarity**
- Return top  $k$  documents

How can we do  
this efficiently?

# First, we need to quantify the similarity of sets: **Jaccard similarity**

- Items are represented as **sets**, e.g.:
  - $A = \{\text{words contained in document } A\}$
  - $B = \{\text{users who interacted with item } B\}$
- **Jaccard similarity** is the ratio
$$J(A, B) = |A \cap B| / |A \cup B|$$
- $D(A, B) = 1 - J(A, B)$  is the **Jaccard distance metric**



# But scale could pose problems:

- Size of collection ( $N$ ) – at least  $N$  comparisons needed
- Dimensionality of representation (could be high  $k$ )
- More generally: complexity of computing similarity
- ***Can we do better (=faster) than brute force search?***

# Instead: Approximate search as a first pass/filter

- Use a **fast method** to identify  $n \ll N$  **candidate nearest neighbors**
- Use **true similarity** on **candidate set** to discard any false positives
- This is common sense/typical for many applications / outside of big data
- Within Data Science, a **fast method** for similarity search of documents usually requires some kind of data structure
  - Search time should be strictly sub-linear:  $n \sim o(N)$
  - e.g.  $\log(N)$  or  $\sqrt{N}$

# Approximate search as a preliminary filter is just common sense

- For instance, imagine a large tech company (or an academic department, for that matter) receives many applications (e.g. 1k) for an advertised position.
- That's a good problem to have for the people doing the hiring, as sample size is large – an ideal candidate is probably in the set of applications, but:
- The ideal candidate will be a good match on many dimensions, which has to be revealed by a multitude of interviews (=many perspectives).
- Is it at all feasible to put all 1k applicants through this process?
- Takes too long / is too much work for one interviewer or committee.
- One solution: Hire more interviewers and interview the candidates in parallel.
- Another solution: Filter by scanning resumes, then only interview (far fewer) candidates that are *\*likely\** a good match.
- Note: This is an approximate search. It does not guarantee success – you might be missing good candidates and also have false positives.

# MinHash

[Broder, 1997]

# Before we go into the details of minHash

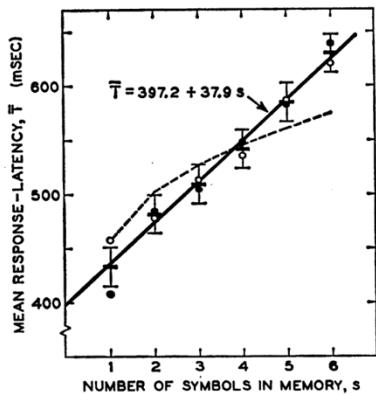
- Some necessary review:
- What is a hash function?
- A function that takes any input and returns fixed-size bytes.
- What is a hash collision?
- When two different inputs yield the same output hash value.
- Example: The digit root (DR) function
- $\text{DR}(12) = 3, \text{DR}(15) = 6, \text{DR}(18) = 9, \text{DR}(21) = 3$
- Digit root of multiples of 9 \*always\* collide (in base 10):
- $\text{DR}(9) = \text{DR}(18) = \text{DR}(27) = \text{DR}(36) = \text{DR}(45) = \text{DR}(54) = \dots = 9$

# The problem with similarity search of large collections, e.g. for plagiarism detection

## Serial scans are inherently slow

### High-Speed Scanning in Human Memory

**Abstract.** When subjects judge whether a test symbol is contained in a short memorized sequence of symbols, their mean reaction-time increases linearly with the length of the sequence. The linearity and slope of the function imply the existence of an internal serial-comparison process whose average rate is between 25 and 30 symbols per second.



the selection of a response requires the use of information that is in memory, the latency of the response will reveal something about the process by which the series varied at random from trial to trial. There was a two-second delay, a warning signal, and then the test digit. As soon as one of the levers

Results are shown in Fig. 1. Linear regression accounts for 99.4 percent of the variance of the overall mean response-latencies (6). The slope of the fitted line is  $37.9 \pm 3.8$  msec per symbol (7); its zero intercept is  $397.2 \pm 19.3$  msec. Lines fitted separately to the mean latencies of positive and negative responses differ in slope by  $9.6 \pm 2.3$  msec per symbol. The difference is attributable primarily to the fact that for  $s = 1$ , positive responses were  $50.0 \pm 20.1$  msec faster than negative responses. Lines fitted to the data for

is retrieved. Of particular interest is the study of retrieval of a number of elements in the response latency. It is found that subjects memorize a short list of symbols in memory. On trials requiring negative responses,  $s$  comparisons must be made. If positive responses were initiated as soon as a match had occurred (as in a self-terminating search), the mean number of comparisons on positive trials would be  $(s + 1)/2$  rather than  $s$ . The latency function for positive responses would then have half the slope of the function for negative responses. The null hypothesis is rejected at the 0.05 level. On each trial of the experiment, subject (4) saw a random set of six different symbols presented simultaneously at a fixed locus. The length,  $s$ , of the stimulus set was varied at random from trial to trial. In experiment 1, the set of symbols associated with the positive response changed from trial to trial. In contrast to this varied-set procedure, a fixed-set procedure was used in experiment 2. In each of three parts of the session, a set of digits for which the

## Many pairwise comparisons are needed

N (documents)	# pairwise comparisons
2	1
10	45
100	4950
1000	499,500
1e4	49.995e6
1e5	49.9995e8
1e6	49.9999e10

# How does MinHash work?

Note:  $\pi$  is not 3.14159... here.  
They just needed a Greek letter

- Fix a random ordering  $\pi$  of the items (items = words)
- Here is a table of set memberships
- For each set  $S$ , its hash is:

$$h(S | \pi) = \min \{k \mid \pi(k) \in S\}$$

- The index of the first (permuted) item belonging to set  $S$

$\pi(k)$		Doc 1	Doc 2	Doc 3	Doc 4	...
1	Item 3	1	0	0	0	
2	Item 75	0	0	1	0	
3	Item 21	0	1	1	0	
4	Item 1	0	0	1	0	
5	Item 2004	0	1	0	1	

$$h(\text{Doc 1} | \pi) = 1$$

$$h(\text{Doc 2} | \pi) = 3$$

$$h(\text{Doc 3} | \pi) = 2$$

$$h(\text{Doc 4} | \pi) = 5$$

# A specific example of permutation indexing in practice

A = {"T.rex", "Stegosaurus", "PDP-11"}

B = {"Apples", "Bananas", "Pine cones"}

C = {"T.rex", "Bananas", "Penguins"}

D = {"Apples", "Turtles", "Pine cones"}

$\pi(k)$		A	B	C	D
1	PDP-11	1	0	0	0
2	Penguins	0	0	1	0
3	Pine cones	0	1	0	1
4	Turtles	0	0	0	1
5	Apples	0	1	0	1
6	T.rex	1	0	1	0
7	Bananas	0	1	1	0
8	Stegosaurus	1	0	0	0

# A specific example of permutation indexing in practice

$$A = \{"T.rex", "Stegosaurus", "PDP-11"\} \rightarrow h(A|\pi) = 1$$

$$B = \{"Apples", "Bananas", "Pine cones"\} \rightarrow h(B|\pi) = 3$$

$$C = \{"T.rex", "Bananas", "Penguins"\} \rightarrow h(C|\pi) = 2$$

$$D = \{"Apples", "Turtles", "Pine cones"\}. \rightarrow h(D|\pi) = 3$$

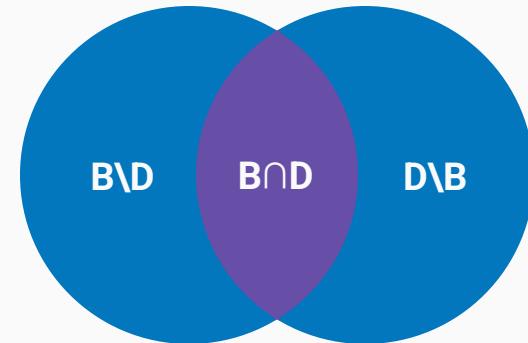
**Hash collision** is more likely when sets overlap.

Let's analyze this more formally!

$\pi(k)$		A	B	C	D
1	PDP-11	1	0	0	0
2	Penguins	0	0	1	0
3	Pine cones	0	1	0	1
4	Turtles	0	0	0	1
5	Apples	0	1	0	1
6	T.rex	1	0	1	0
7	Bananas	0	1	1	0
8	Stegosaurus	1	0	0	0

$$P[h(S_1) = h(S_2)] = \text{Jaccard}(S_1, S_2)$$

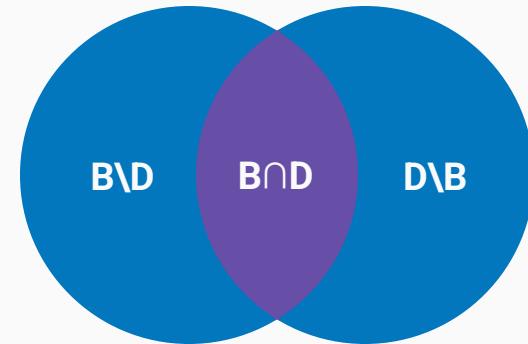
- For two sets  $S_1$  and  $S_2$ , there are three types of rows:
  - Type 1:  $\pi(k) \in S_1 \cap S_2$
  - Type 2:  $\pi(k) \in S_1 \Delta S_2$
  - Type 3:  $\pi(k) \notin S_1 \cup S_2$
- Insight:**  
**Collision**  $\Leftrightarrow$  type 1 row occurs before all type 2 rows



$\pi(k)$	A	B	C	D
1 PDP-11	1	0	0	0
2 Penguins	0	0	1	0
3 Pine cones	0	1	0	1
4 Turtles	0	0	0	1
5 Apples	0	1	0	1
6 T.rex	1	0	1	0
7 Bananas	0	1	1	0
8 Stegosaurus	1	0	0	0

$$P[h(S_1) = h(S_2)] = \text{Jaccard}(S_1, S_2)$$

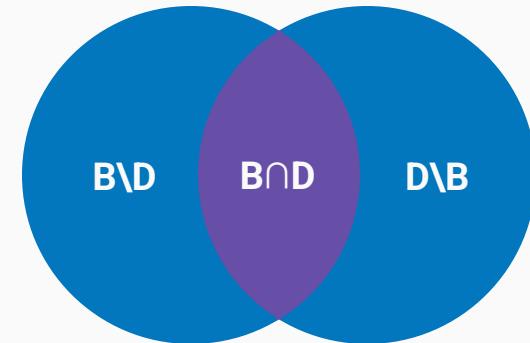
- For two sets  $S_1$  and  $S_2$ , there are three types of rows:
  - Type 1:  $\pi(k) \in S_1 \cap S_2$
  - Type 2:  $\pi(k) \in S_1 \Delta S_2$
  - Type 3:  $\pi(k) \notin S_1 \cup S_2$
- Collision**  $\Leftrightarrow$  type 1 row occurs before all type 2 rows
- $P[\text{Collision}] = (\# \text{ Type 1}) / (\# \text{ Type 1} + \# \text{ Type 2})$



$\pi(k)$	A	B	C	D
1 PDP-11	1	0	0	0
2 Penguins	0	0	1	0
3 Pine cones	0	1	0	1
4 Turtles	0	0	0	1
5 Apples	0	1	0	1
6 T.rex	1	0	1	0
7 Bananas	0	1	1	0
8 Stegosaurus	1	0	0	0

$$P[h(S_1) = h(S_2)] = \text{Jaccard}(S_1, S_2)$$

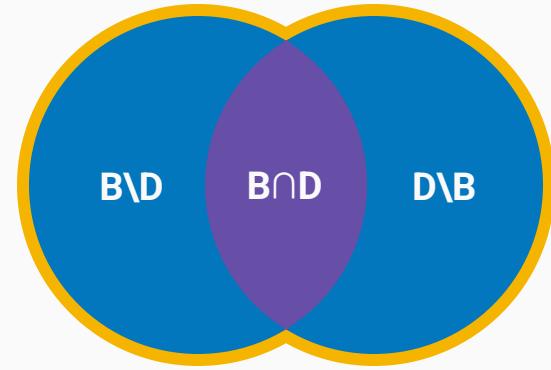
- For two sets  $S_1$  and  $S_2$ , there are three types of rows:
  - Type 1:  $\pi(k) \in S_1 \cap S_2$
  - Type 2:  $\pi(k) \in S_1 \Delta S_2$
  - Type 3:  $\pi(k) \notin S_1 \cup S_2$
- Collision**  $\Leftrightarrow$  type 1 row occurs before all type 2 rows
- $P[\text{Collision}] = (\# \text{ Type 1}) / (\# \text{ Type 1} + \# \text{ Type 2})$   
 $= |S_1 \cap S_2| / (|S_1 \cap S_2| + |S_1 \Delta S_2|)$



$\pi(k)$	A	B	C	D
1 PDP-11	1	0	0	0
2 Penguins	0	0	1	0
3 Pine cones	0	1	0	1
4 Turtles	0	0	0	1
5 Apples	0	1	0	1
6 T.rex	1	0	1	0
7 Bananas	0	1	1	0
8 Stegosaurus	1	0	0	0

$$P[h(S_1) = h(S_2)] = \text{Jaccard}(S_1, S_2)$$

- For two sets  $S_1$  and  $S_2$ , there are three types of rows:
  - Type 1:  $\pi(k) \in S_1 \cap S_2$
  - Type 2:  $\pi(k) \in S_1 \Delta S_2$
  - Type 3:  $\pi(k) \notin S_1 \cup S_2$
- Collision**  $\Leftrightarrow$  type 1 row occurs before all type 2 rows
- $P[\text{Collision}] = (\# \text{ Type 1}) / (\# \text{ Type 1} + \# \text{ Type 2})$   
 $= |S_1 \cap S_2| / (|S_1 \cap S_2| + |S_1 \Delta S_2|)$   
 $= |S_1 \cap S_2| / (|S_1 \cup S_2|)$   
 $= J(S_1, S_2)$  ■



$\pi(k)$	A	B	C	D
1 PDP-11	1	0	0	0
2 Penguins	0	0	1	0
3 Pine cones	0	1	0	1
4 Turtles	0	0	0	1
5 Apples	0	1	0	1
6 T.rex	1	0	1	0
7 Bananas	0	1	1	0
8 Stegosaurus	1	0	0	0

## Monte Carlo approximation of all possible permutations

- B and D had  $P[\text{collision}] = \frac{1}{2}$  for a single permutation  $\pi$
- But we want the **probability of collision** over all choices of  $\pi$

Table of set memberships of one particular permutation

- Idea: Table of minhash signatures across permutations

- Generate  $m$  random permutations  $\pi_1, \pi_2, \dots, \pi_m$
- Count hash collisions between A and B over all  $\pi_i$ 's
- $J(A, B) \approx \# \text{ collisions} / m$

$\pi(k)$	A	B	C	D
1 PDP-11	1	0	0	0
2 Penguins	0	0	1	0
3 Pine cones	0	1	0	1
4 Turtles	0	0	0	1
5 Apples	0	1	0	1
6 T.rex	1	0	1	0
7 Bananas	0	1	1	0
8 Stegosaurus	1	0	0	0



$h(S   \pi_i)$	A	B	C	D
$\pi_1$	7	1	2	1
$\pi_2$	2	1	2	5
$\pi_3$	4	5	3	6
$\pi_4$	9	5	1	1
$\pi_5$	3	2	6	2
...				

MinHash signatures

But full permutations or large sets is still costly  
Can we approximate the permutations too?

- Computing and storing random permutations of large collections is not practical, as it is simply too costly
- Instead, we can replace **permutations  $\pi_i$**  with **hashes  $H_i$** 
  - A permutation is a **perfect hash**: distinct elements cannot collide
  - Approximate this by an **imperfect hash**: distinct elements **may collide**
- As long as  $H_i$  are unlikely to collide, this can still work
- Note: This is an approximation to an approximation (2<sup>nd</sup> order).

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	$\infty$	$\infty$	$\infty$	$\infty$
$H_2$	$\infty$	$\infty$	$\infty$	$\infty$

Signature table is initialized to infinity for each entry

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0 $\infty$	$\infty$	$\infty$	$\infty$
$H_2$	0 $\infty$	$\infty$	$\infty$	$\infty$

A,  $H_1$ :  $0 < \infty \rightarrow$  update  
A,  $H_2$ :  $0 < \infty \rightarrow$  update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	$\infty$	1 $\infty$	$\infty$
$H_2$	0	$\infty$	2 $\infty$	$\infty$

C,  $H_1$ :  $1 < \infty \rightarrow$  update  
C,  $H_2$ :  $2 < \infty \rightarrow$  update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	2 $\infty$	1	2 $\infty$
$H_2$	0	4 $\infty$	2	4 $\infty$

B,  $H_1$ :  $2 < \infty \rightarrow$  update  
B,  $H_2$ :  $4 < \infty \rightarrow$  update  
D,  $H_1$ :  $2 < \infty \rightarrow$  update  
D,  $H_2$ :  $4 < \infty \rightarrow$  update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	2	1	2
$H_2$	0	4	2	0 4

D,  $H_1$ :  $3 > 2 \rightarrow$  no update  
D,  $H_2$ :  $0 < 4 \rightarrow$  update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	0 2	1	0 2
$H_2$	0	1 3	2	0

B,  $H_1$ :  $0 < 2 \rightarrow$  update  
 B,  $H_2$ :  $1 < 3 \rightarrow$  update  
 D,  $H_1$ :  $0 < 2 \rightarrow$  update  
 D,  $H_2$ :  $1 > 0 \rightarrow$  no update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0

A,  $H_1$ :  $1 > 0 \rightarrow$  no update  
A,  $H_2$ :  $3 > 0 \rightarrow$  no update  
C,  $H_1$ :  $1 = 1 \rightarrow$  no update  
C,  $H_2$ :  $3 > 2 \rightarrow$  no update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0

B,  $H_1$ :  $2 > 0 \rightarrow$  no update  
B,  $H_2$ :  $5 > 1 \rightarrow$  no update  
C,  $H_1$ :  $2 > 1 \rightarrow$  no update  
C,  $H_2$ :  $5 > 2 \rightarrow$  no update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0

A,  $H_1$ :  $3 > 0 \rightarrow$  no update  
A,  $H_2$ :  $1 > 0 \rightarrow$  no update

# Computing minhash signatures with hash functions (approximate minhash)

$x$	$H_1(x)$	$H_2(x)$	A	B	C	D
PDP-11	0	0	1			
Penguins	1	2			1	
Pine cones	2	4		1		1
Turtles	3	0				1
Apples	0	1		1		1
T.rex	1	3	1		1	
Bananas	2	5		1	1	
Stegosaurus	3	1	1			

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0



Collisions:

- $H_1$ : A ≡ B ≡ D ≢ C
- $H_2$ : A ≡ D ≢ B ≢ C

# Big picture: Similarity search and minHash

- Similarity search is straightforward – just compare the similarity of documents in a collection with a metric like Jaccard similarity.
- However, this is too costly for large collections, so an approximation that identifies candidates is indicated, we can then compute full similarity for candidate subset.
- To identify candidates, we can use minHash, as the probability of a hash collision corresponds to the Jaccard similarity.
- Conceptually, we can use random permutations, as the number of collisions in the table of minHash signatures corresponds to the Jaccard similarity (scaled by the number of permutations we do) to identify a candidate set.
- There is a 2<sup>nd</sup> layer. In addition to using minHash, we can also use hash functions as a proxy for permutations, as full permutations are too costly for large collections.
- AFTER minHash has identified a – much reduced - candidate set, we can do the full similarity search (as it will be a small data problem at that point).

# Failure modes of MinHash

- Permutation MinHash:
  - **Collisions** are **more likely** when a **small set of items** are shared across **many documents**
  - ⇒ “Stop-words” can be “deadly”! “*The*”, “*and*”, “*or*”, etc...
- **Hashing approximation** only makes things **worse**
  - Two distinct items can now hash to the same value
  - Collision probability only increases with the hashing approximation
  - **High collision likelihood** ⇒ **large candidate sets** ⇒ **slow retrieval**

Key / solution: Keep the candidate set as small as possible, but that might lower recall

Now:

*Can we reduce the size of candidate sets,  
but maintain high recall?*

# Locality sensitive hashing

[Indyk and Motwani, 1998]

[Charikar 2002]

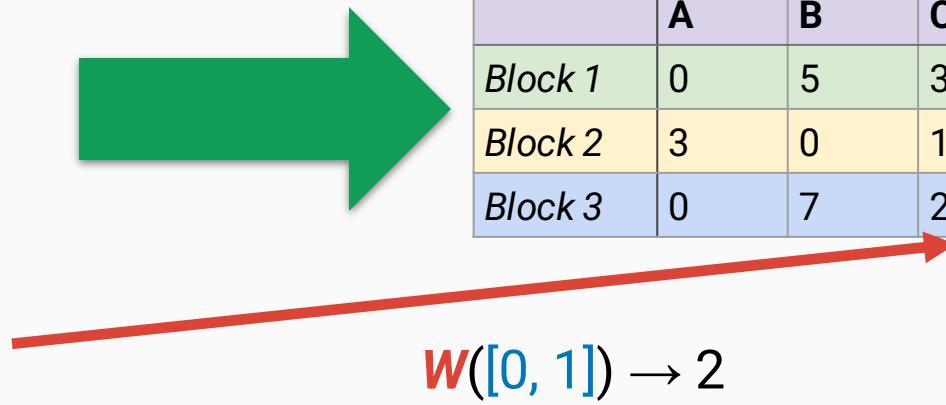
# LSH: Improving the efficiency of MinHash

- Traditional hash functions scatter data “**randomly**”
  - Probability of collision is **independent of similarity** between inputs
- **The big idea:** Locality-sensitive hashes have a higher probability of collision for inputs that are ***near each other*** (*relative to inputs that are far*)
- LSH has a huge literature, this part will be introductory

# LSH + MinHash

- Carve signature matrix into  $b$  blocks of  $R$  rows each
- Hash each sub-column with a standard (non-local) hash function  $W$ 
  - Pick  $W$  such that collisions are rare for non-identical inputs
- Candidate set = items that collide in any row block

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

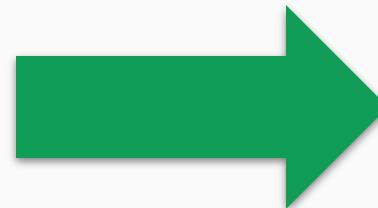


# LSH + MinHash

- Carve signature matrix into  $b$  blocks of  $R$  rows each
- Hash each submatrix
- Hash each submatrix
  - Pick  $W$  such that
- Candidate set = items that collide in *any* row block

**What's the probability that we have at least one block where the hashes of all rows match?**

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1



	A	B	C	D
Block 1	0	5	3	0
Block 2	3	0	1	3
Block 3	0	7	2	2

$$W([0, 1]) \rightarrow 2$$

# LSH vs direct MinHash analysis

- MinHash:
  - $P[\text{single row collision}] = J(A, B) = j$
- LSH:
  - $P[\text{all } R \text{ rows in a block collide}] = j^R$

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

	A	B	C	D
<i>Block 1</i>	0	5	3	0
<i>Block 2</i>	3	0	1	3
<i>Block 3</i>	0	7	2	2

# LSH vs direct MinHash analysis

- MinHash:
  - $P[\text{single row collision}] = J(A, B) = j$
- LSH:
  - $P[\text{all } R \text{ rows in a block collide}] = j^R$
  - $P[\text{at least one non-collision in a block}] = 1 - j^R$

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

	A	B	C	D
<i>Block 1</i>	0	5	3	0
<i>Block 2</i>	3	0	1	3
<i>Block 3</i>	0	7	2	2

# LSH vs direct MinHash analysis

- MinHash:

- $P[\text{single row collision}] = J(A, B) = j$

- LSH:

- $P[\text{all } R \text{ rows in a block collide}] = j^R$
  - $P[\text{at least one non-collision in a block}] = 1 - j^R$
  - $P[\text{at least one non-collision in all } b \text{ blocks}] = (1 - j^R)^b$

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

	A	B	C	D
<i>Block 1</i>	0	5	3	0
<i>Block 2</i>	3	0	1	3
<i>Block 3</i>	0	7	2	2

# LSH vs direct MinHash analysis

- MinHash:

- $P[\text{single row collision}] = J(A, B) = j$

- LSH:

- $P[\text{all } R \text{ rows in a block collide}] = j^R$
  - $P[\text{at least one non-collision in a block}] = 1 - j^R$
  - $P[\text{at least one non-collision in all } b \text{ blocks}] = (1 - j^R)^b$
  - $P[\text{at least one block collides on all rows}] = 1 - (1 - j^R)^b$

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

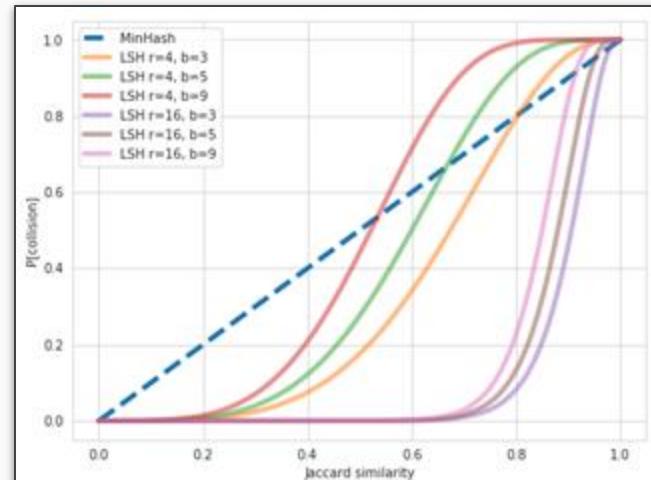
	A	B	C	D
<i>Block 1</i>	0	5	3	0
<i>Block 2</i>	3	0	1	3
<i>Block 3</i>	0	7	2	2

# LSH vs direct MinHash analysis

- MinHash:
  - $P[\text{single row collision}] = J(A, B) = j$
- LSH:
  - $P[\text{all } R \text{ rows in a block collide}] = j^R$
  - $P[\text{at least one non-collision in a block}] = 1 - j^R$
  - $P[\text{at least one non-collision in all } b \text{ blocks}] = (1 - j^R)^b$
  - $P[\text{at least one block collides on all rows}] = 1 - (1 - j^R)^b$

	A	B	C	D
$H_1$	0	0	1	0
$H_2$	0	1	2	0
$H_3$	1	2	0	1
$H_4$	0	1	0	0
$H_5$	2	2	0	0
$H_6$	1	2	1	1

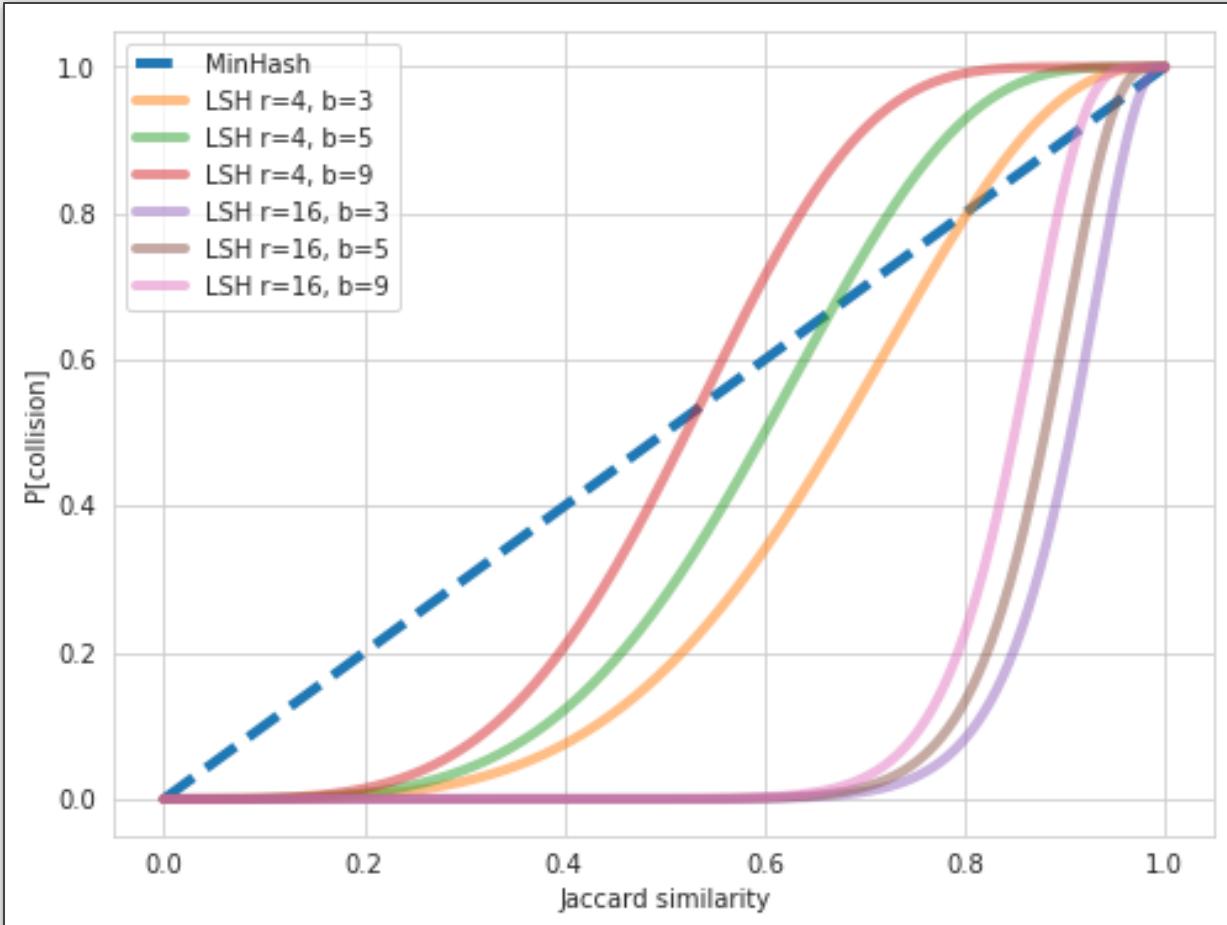
	A	B	C	D
Block 1	0	5	3	0
Block 2	3	0	1	3
Block 3	0	7	2	2



## Result:

Collisions are **more likely** for high Jaccard similarity and **less likely** for low Jaccard similarity

# Tradeoffs everywhere...



Beyond simple sets:

- 1) Bags
- 2) Spatial similarity search  
(as in a vector database)

What if you don't just want to consider whether an item appears (in a set), but also how often it does so (in a multi-set/bag):

## Ruzicka similarity

[Chen, Philbin, Zisserman 2008]

- Idea: reduce bags to sets by uniquely identifying each repetition  $\min(x_1, y_1) = \min(1, 2) = 1$

$\{\text{dog}\}$	$\rightarrow \{\text{dog}_1\}$	$\mathbf{A} \quad \mathbf{B} \quad \mathbf{C} \quad \mathbf{D}$	$\max(x_1, y_1) = \max(1, 2) = 2$
$\{\text{dog}, \text{dog}\}$	$\rightarrow \{\text{dog}_1, \text{dog}_2\}$	$X = [1, 3, 5, 7]$	$\min(x_2, y_2) = \min(3, 3) = 3$
$\{\text{dog}, \text{dog}, \text{dog}\}$	$\rightarrow \{\text{dog}_1, \text{dog}_2, \text{dog}_3\}$	$Y = [2, 3, 1, 6]$	$\max(x_2, y_2) = \max(3, 3) = 3$

- Jaccard on expanded sets = **Ruzicka similarity** on original bags  $\min(x_3, y_3) = \min(5, 1) = 1$

$$R(\mathbf{A}, \mathbf{B}) = \frac{\sum_i \min(\mathbf{A}[i], \mathbf{B}[i])}{\sum_j \max(\mathbf{A}[j], \mathbf{B}[j])}$$

$\Sigma(\min(x_i, y_i)) = 1 + 3 + 1 + 6 = 11$        $\max(x_3, y_3) = \max(5, 1) = 5$   
 $\Sigma(\max(x_i, y_i)) = 2 + 3 + 5 + 7 = 17$        $\min(x_4, y_4) = \min(7, 6) = 6$   
 $R(X, Y) = 11 / 17 \approx 0.65$        $\max(x_4, y_4) = \max(7, 6) = 7$

# Cosine similarity is often a suitable metric to compare documents (and other high-dimensional objects)

- For instance, one could express documents in terms of the magnitude of their embedding dimensions.
- Cosine similarity conceptualizes the similarity of two vectors  $\mathbf{a}$  and  $\mathbf{b}$  in terms of the angle  $\theta$  between them, regardless of their length.
- This makes intuitive sense:

$$\text{cosine similarity} = \cos(\theta) = \frac{\mathbf{a}^T \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \frac{\mathbf{a}^T \mathbf{b}}{\sqrt{\mathbf{a}^T \mathbf{a}} \sqrt{\mathbf{b}^T \mathbf{b}}}$$

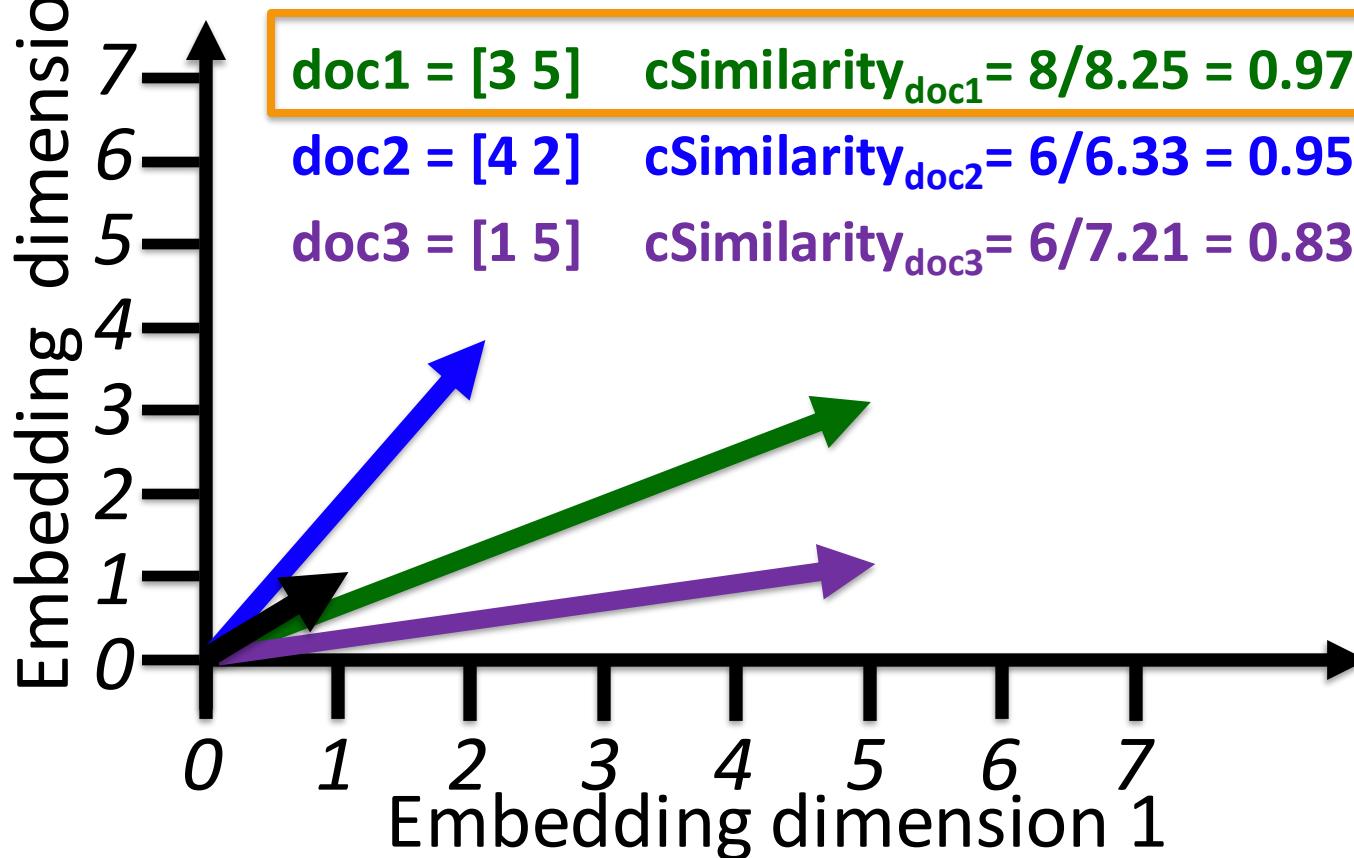
$$\cos(0) = 1$$

$$\cos(90) = 0$$

$$\cos(180) = -1$$

# Determining relevance with cosine similarity

$\text{doc}_0 = [1 \ 1]$

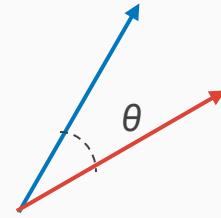


# LSH for cosine similarity

[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$



# LSH for cosine similarity

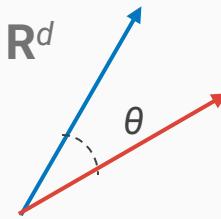
[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  randomly but uniformly from the unit sphere in  $\mathbb{R}^d$

- $h_w(x) = 1$  if  $w^T x \geq 0$   
 $= -1$  if  $w^T x < 0$



# LSH for cosine similarity

[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

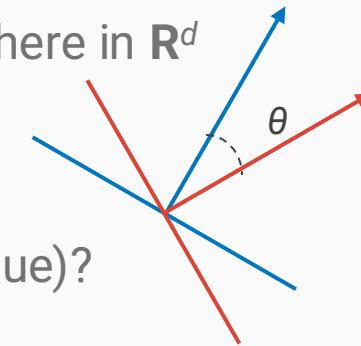
$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  randomly but uniformly from the unit sphere in  $\mathbb{R}^d$

- $h_w(x) = 1$  if  $w^T x \geq 0$   
 $= -1$  if  $w^T x < 0$

- What's the probability of a collision (both same hash value)?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)]$



# LSH for cosine similarity

[Charikar 2002]

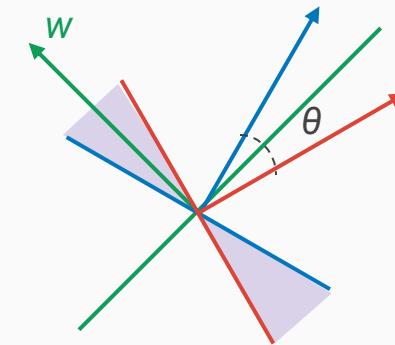
- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  uniformly from the sphere in  $\mathbb{R}^d$ 
  - $h_w(x) = 1$  if  $w^T x \geq 0$
  - $= -1$  if  $w^T x < 0$

- What's the probability of collision?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)] = 1 - P[w \text{ in shaded region}]$



# LSH for cosine similarity

[Charikar 2002]

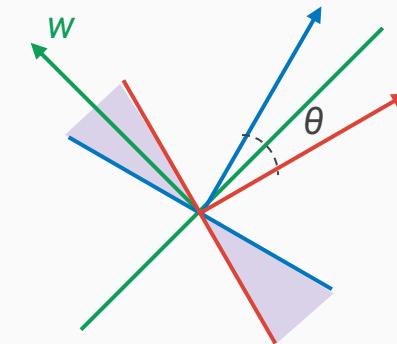
- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  uniformly from the sphere in  $\mathbb{R}^d$ 
  - $h_w(x) = 1$  if  $w^T x \geq 0$
  - $= -1$  if  $w^T x < 0$

- What's the probability of collision?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)]$   
 $= 1 - P[w \text{ in shaded region}]$   
 $= 1 - 2 \cdot |\theta| / 2\pi$   
 $= 1 - |\theta|/\pi$



Not exactly  $\cos(\theta)$ , but monotonically decreasing with  $|\theta| \Rightarrow$  same rank-ordering

# Multiple projections

- $P[\text{No collision} \mid \text{single projection}] = \theta/\pi$
- $P[\text{All projections do not collide} \mid m \text{ projections}] = (\theta/\pi)^m$
- $P[\text{At least one Collision} \mid m \text{ projections}] = 1 - (\theta/\pi)^m$

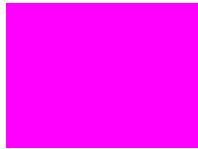
# Wrap-up

- Similarity search is straightforward, but scale can be overwhelming
- MinHash is simple, but low-precision
- LSH can improve precision, while retaining recall
- This approach generalizes to spatial similarity metrics (beyond sets)

# Outlook

- This week:
  - Release of HW4
  - Release of Capstone project
- Next week:
  - Graph search

**Q & R**



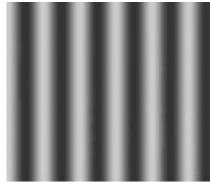
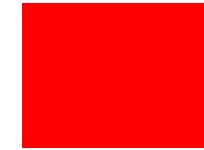
Smallest font



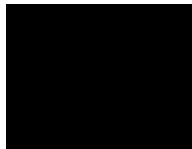
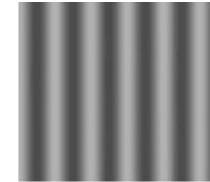
Please turn off and put  
away your cell phone



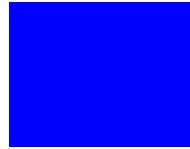
# Calibration slide



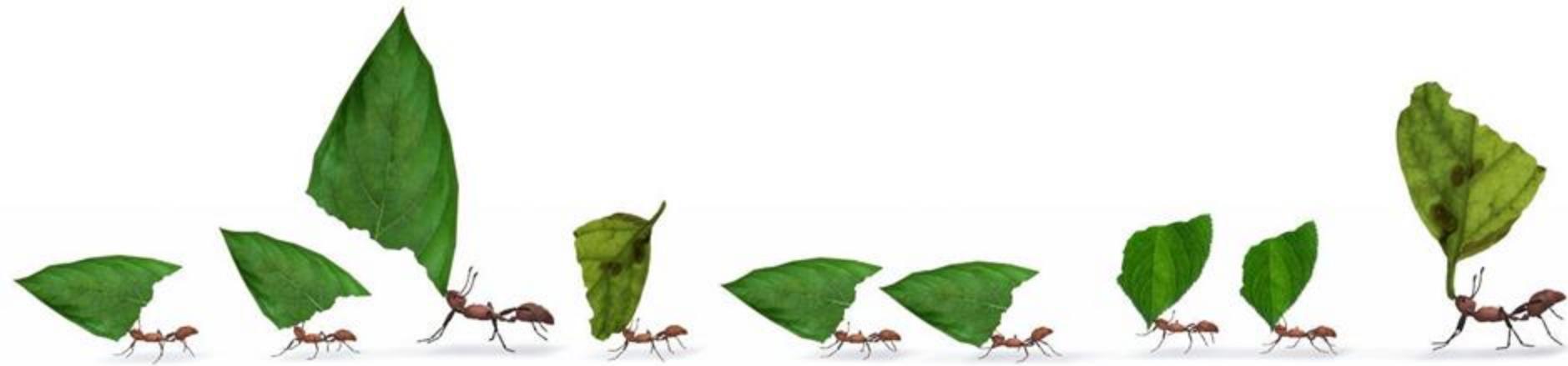
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



# Big Data





# Recommender systems

DS-GA 1004: Big Data



# This week

- Recommender systems
- Collaborative filtering algorithms
- Ranking and evaluation

# Confusion, Doubt, & Struggle: Search

- **Approximate minHash** (using hashes instead of permutations): The point of the permutations in minHash is to get a single integer number (the first time the concept is encountered in the document, given the permutation), then check for collisions.
- But if one is doing that, the ultimate purpose of the permutations is a single integer, which we can get more cheaply from a hash function (even doing a single permutation, if there is a large number of items is expensive). But the hash function doesn't mind a large input.
- **How do different outputs of different Hashes ( $H_1$ ,  $H_2$ , etc.) of the same hash function  $H$  come from?** The hash functions used for this purpose (e.g. "MurmurHash") are families of functions that take parameters. In approximate minhash, we can use random coefficients for these parameters, e.g.  $\text{hash}(x) = (a*x + b) \bmod p$ , where  $x$  is the input,  $a$  and  $b$  are random coefficients and  $p$  is a large prime number. Those come from a seed, which is an input to MurmurHash (in addition to the input).
- If we look for similarity of documents, we have to reckon with "stop words" – words that are commonly used, which will lead to too many collisions for plain vanilla minHash. So precision will be low, defeating the purpose of doing approximate search in the first place.
- One solution: Instead of curating stop-words, use minHash with LSH. We can partition the rows into a number  $b$  of blocks and  $r$  of rows, but there are many tradeoffs between the number of blocks and the number of rows per block we use. If we ask for many rows (only considering it a collision if all rows in a block collide), the probability of random collisions will go way down, so precision will go up. If we increase the number of blocks (counting a collision in any block), recall will go up again.

# Search

Relevance from **content**  
similarity between query  
and document

**Personalization** of  
relevance from  
explicit or implicit  
feedback

# Recommendation

Relevance from  
network **structure**

# Graph algorithms

# Now: From Search to Recommendation

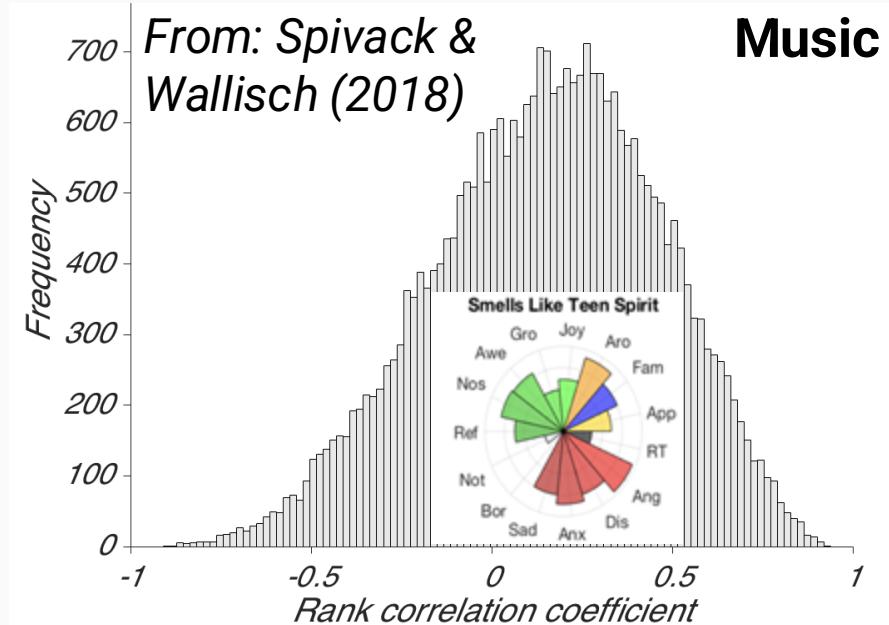
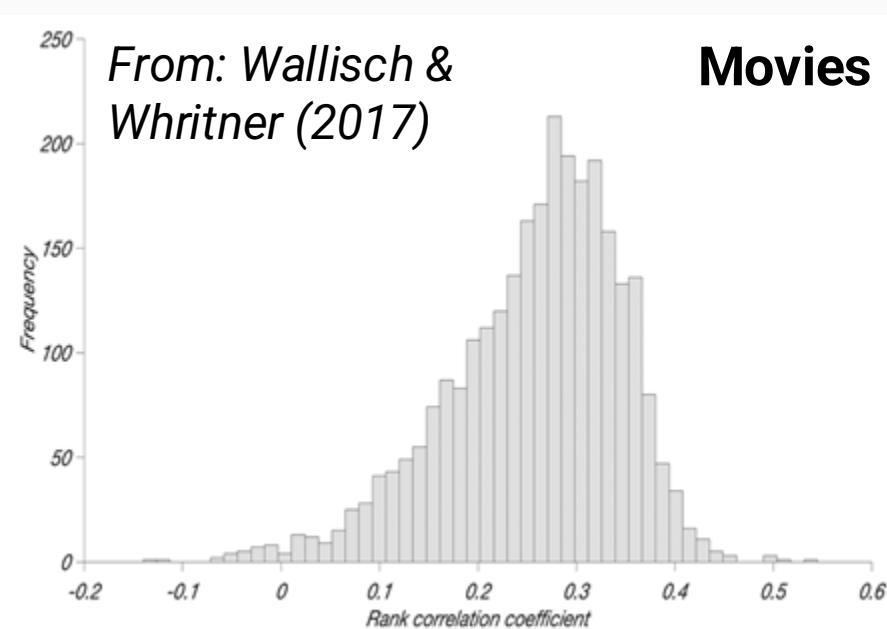
- Early solutions relied on **indexing** and **search**
  - Users must describe what they want
  - Same query by different users gets same results
- **Recommendation** = **search + personalization**
  - A user's past history informs the ranking of results
  - Not everyone has the same preferences
  - Relevance, but personalized...
  - Other users' histories are also informative!



Yahoo.com in April, 1997 [archive.org]

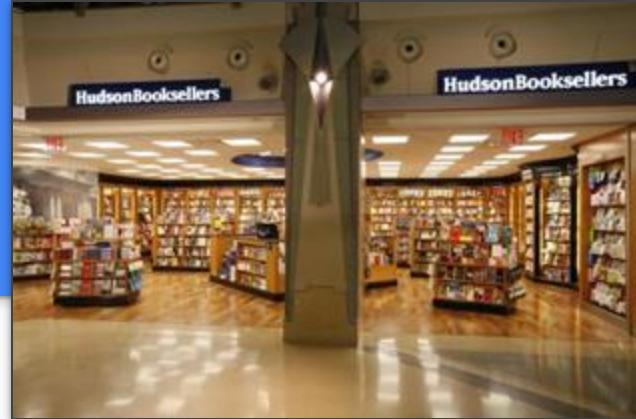
# Why the need for recommendation (personalization)?

Because people are complicated. There is genuine and dramatic diversity in preferences:



# Good timing:

This need coincided with digital affordances in the 21<sup>st</sup> century



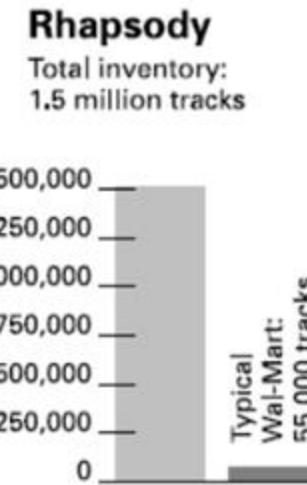
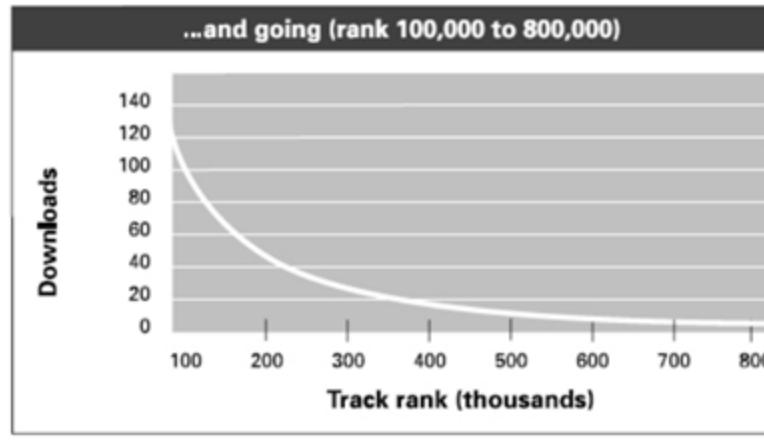
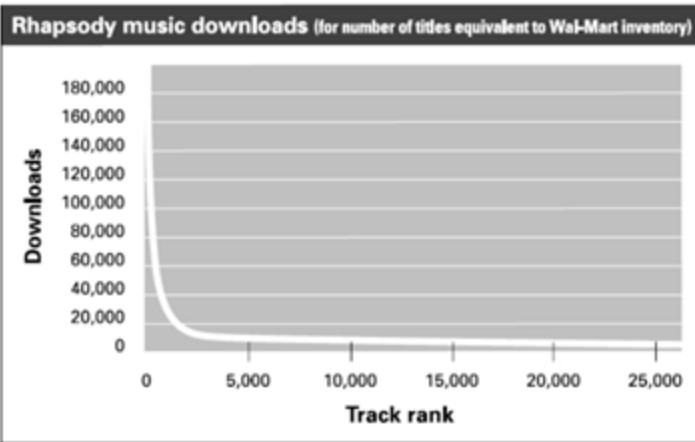
- **Physical objects occupy space**

- Brick-and-mortar shops must satisfy physical constraints
- Curators must prioritize for some notion of utility
- Serving the most customers, maximizing sales/profit, etc.

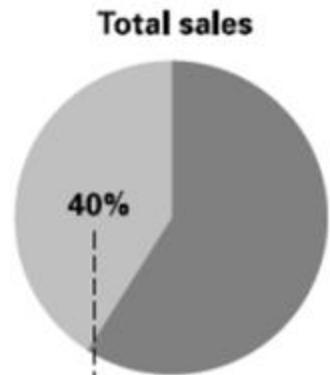
- This is not true for **digital items!**

- The web, e-books, news articles, movies, music, ... take up no physical space
- Without algorithmic curation, this quickly becomes overwhelming

# Contrasting the brick/mortar “short head” with the digital “long tail”



From: Chris Anderson:  
“The long tail” (2006)

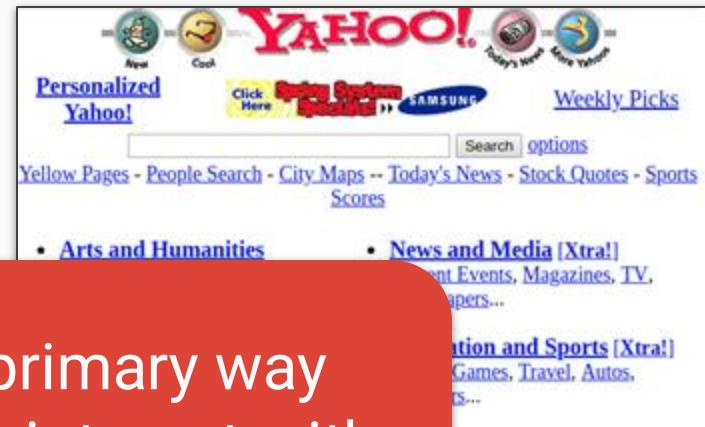


# Search ⇒ Recommendation

- Early solutions relied on **indexing** and **search**
  - Users must describe what they want
  - Same query by different users gets same results
- **Recommendation - search - personalization**

Recommender systems are now the primary way that most people (often unknowingly) interact with large collections!

At this point, >70% of clicks on Youtube and >80% on Netflix result from recommendations (not search or direct links)



# Personalization is critical to recommendation

- Traditional search / information retrieval:
  - Model the **relevance** of an **item** in response to a **query**
- Personalized search / recommendation
  - Same as above, but the model can access a description of the **user**
- To model **relevance** per user, we'll need to record data
  - This is known as **feedback**
- This approach works best if you have big data

And yet, there is a shared signal:  
The baseline tailored recommendation systems  
need to beat: The “popularity based model”

- This should always be the first thing you try.

- 1. Compute the **average utility** for each item

$$P[i] \leftarrow (\sum_u R[u, i]) / |R[:, i]|$$

- 2.  $f$  = list of items ranked by descending  $P[i]$

- AKA: the “**short head**”; **greatest hits**; **Starbucks mix**, ...
- This produces the **same ranking for all users**.  
**Personalization** should improve, but by how much?



# Some good examples

## IMDb Charts

### IMDb Top 250 Movies

As rated by regular IMDb voters.

250 Titles



Sort by **Ranking** ↑

1. **The Shawshank Redemption**  
1994 2h 22m R  
★ 9.3 (2.8M) ⚡ Rate

2. **The Godfather**  
1972 2h 55m R  
★ 9.2 (2M) ⚡ Rate

3. **The Dark Knight**  
2008 2h 32m PG-13  
★ 9.0 (2.8M) ⚡ Rate

4. **The Godfather Part II**  
1974 3h 22m R  
★ 9.0 (1.3M) ⚡ Rate

5. **12 Angry Men**  
1957 1h 36m Approved  
★ 9.0 (840K) ⚡ Rate

6. **Schindler's List**  
1993 3h 15m R  
★ 9.0 (1.4M) ⚡ Rate

7. **The Lord of the Rings: The Return of the King**  
2003 3h 21m PG-13  
★ 9.0 (1.9M) ⚡ Rate

8. **Pulp Fiction**  
1994 2h 34m R  
★ 8.9 (2.2M) ⚡ Rate

9. **The Lord of the Rings: The Fellowship of the Ring**  
2001 2h 58m PG-13  
★ 8.8 (2M) ⚡ Rate

10. **The Good, the Bad and the Ugly**  
1966 2h 58m Approved  
★ 8.8 (79M) ⚡ Rate

## Billboard Hot 100™

WEEK OF APRIL

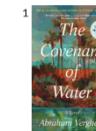
THIS WEEK	AWARD	LAST WEEK	PEAK POS.
	Like That Future, Metro Boomin & Kendrick Lamar	1	1
	Too Sweet Hozier	★	4 2
	Beautiful Things Benson Boone	★	3 2
	Lose Control Teddy Swims	★	5 1
	Texas Hold 'Em Beyoncé	+	2 1
	7 Minute Drill J. Cole	(★)	- 6
	Lovin On Me Jack Harlow	+	8 1
	We Can't Be Friends (Wait For Your L... Ariana Grande	+	9 1
	Type Shit Future, Metro Boomin, Travis Scott & Playboi Carti	+	10 2
	Stick Season Noah Kahan	★	15 10

## B&N Top 100: Bestselling Books

Discover the best books to read that are trending right now.

1 - 20 of 100 results

Viewing availability for Buy Online, Pick up in Store at B&N Union Square Change My Store



**The Covenant of Water (Oprah's Book Club)** (05/02/2024)

by Abraham Verghese

★★★★★

Format

Hardcover

\$24.99 \$32.00

SAVE 22%

Also available as: Large Print, See All >

Availability  
✓ Available Online  
✓ In Stock at My Store  
✓ Store Pickup Available

ADD TO CART



**ONYX STORM (Deluxe Limited Edition)** (01/21/2025)

by Rebecca Yarros

★★★★★

Format

Hardcover

\$32.99

Also available as: Hardcover, See All >

Availability  
✓ Pre-order Now

PRE-ORDER



**Jujutsu Kaisen, Vol. 22** (04/16/2024)

by Gege Akutami

★★★★★

Format

Paperback

\$11.99

Availability  
✓ Available Online  
✓ In Stock at My Store  
✓ Store Pickup Available

ADD TO CART



**The Women: A Novel** (02/06/2024)

by Kristin Hannah

★★★★★

Format

Hardcover

\$21.00 \$30.00

SAVE 30%

Also available as: Audio CD, See All >

Availability  
✓ Available Online  
✓ In Stock at My Store  
✓ Store Pickup Available

ADD TO CART



**Funny Story** (04/23/2024)

by Emily Henry

★★★★★

Format

Hardcover

\$24.65 \$29.00

SAVE 15%

Also available as: Signed Book, Large Print, Audio CD, See All >

Availability  
✓ Pre-order Now

PRE-ORDER

# Personalized recommendations can be based on implicit and explicit feedback

## Explicit feedback

- Examples:
  - \$\$\$ (Purchases)
  - ★★★☆☆
  - 🌟👎
  - +Subscribe / Conversion
- **Strong signal** (+/-)
- **Relatively rare**: users will not always provide explicit feedback - concerns about selection bias.

# Explicit vs. implicit feedback: Tradeoffs

## Explicit feedback

- Examples:
  - \$\$\$ (Purchases)
  - ★★★☆☆
  - 🤗👎
  - +Subscribe / Conversion
- **Strong signal** (+/-)
- **Relatively rare**: users will not always provide explicit feedback - concerns about selection bias.

## Implicit feedback

- Examples:
  - Click-through
  - Downloads
  - Play counts
  - Abandonment / skipping
- **Weak, ambiguous signal** (+ only, usually)
- **Much more abundant** than explicit feedback

# How is feedback obtained?

- **Explicit feedback** makes sense when

Time/effort to rate item is **much less** than time to consume the item!

- **Examples:** movies, books, restaurants, courses, ...
- **Implicit feedback** is currently more commonly used → Big Data
  - **Examples:** Songs, news articles, short videos, search engine results, ...

# We can/will use both implicit and explicit feedback to personalize recommendations. But first: Improving the popularity baseline without tailoring

- Which would you rather have?
  - Item 1: **1000 ratings**, avg = ★★★★☆
  - Item 2: **1 rating**, avg = ★★★★☆
- Few **interactions**  $|R[:, i]| \Rightarrow$  **unstable estimates**  $P[i]$
- **Quick fix 1:** discard items with too few ratings
- **Quick fix 2:** use a **prior**:
$$P[i] \leftarrow (\sum_u R[u, i]) / (|R[:, i]| + \beta)$$
  - $\beta$ : A hyperparameter that “damps” all ratings, but impacts low n averages more
  - AKA “**pseudo-counts**” (equivalent to adding extra observations with R=0)

# Global, item, and user “bias”

- Model each interaction as a combination of **global**, **item**, and **user** terms:

$$R[u, i] \approx \mu + b[i] + b[u]$$

# Global, item, and user “bias”

- Model each interaction as a combination of **global**, **item**, and **user** terms:

$$R[u, i] \approx \mu + b[i] + b[u]$$

- The average rating over all interactions:

$$\mu = (\sum_{u,i} R[u, i]) / (|R| + \beta_g)$$

# Global, item, and user “bias”

- Model each interaction as a combination of **global**, **item**, and **user** terms:

$$R[u, i] \approx \mu + b[i] + b[u]$$

- The average rating over all interactions:

$$\mu = (\sum_{u,i} R[u, i]) / (|R| + \beta_g)$$

- Average (**users**) difference from  $\mu$  for item  $i$ :

$$b[i] = (\sum_u R[u, i] - \mu) / (|R[:, i]| + \beta_i)$$

# Global, item, and user “bias”

- Model each interaction as a combination of **global**, **item**, and **user** terms:

$$R[u, i] \approx \mu + b[i] + b[u]$$

- The average rating over all interactions:

$$\mu = (\sum_{u,i} R[u, i]) / (|R| + \beta_g)$$

- Average (**users**) difference from  $\mu$  for item  $i$ :

$$b[i] = (\sum_u R[u, i] - \mu) / (|R[:, i]| + \beta_i)$$

- Average (**items**) difference from  $\mu + b[i]$  for user  $u$ :

$$b[u] = (\sum_i R[u, i] - \mu - b[i]) / (|R[u]| + \beta_u)$$

# Recommending with the “bias model”

- $R[u, i] \approx \mu + b[i] + b[u]$  lets us estimate all interactions
- Predictions for user  $u$ : sort items  $i$  by descending  $\mu + b[i] + b[u]$
- $\mu$  and  $b[u]$  are constant in this prediction, since  $u$  is fixed.
- Equivalently, we can sort by the item bias  $b[i]$
- This isn't any more powerful than the original popularity model (every user gets the same item bias list)
  - but it is a lot more interpretable (pure item (“bias”))!

# Going beyond popularity baselines and bias models with “Collaborative filtering”



# Collaborative filtering: The task / challenge

- **Utility matrix ( $R$ ):** feedback for sparsely observed interactions

Items

Users	Items				
	1	1	0	0	1
1	1		1		
	1	0			

- **Task:** predict the missing entries  $\rightarrow R_{ui} \approx f(\text{User} = u, \text{Item} = i)$
- **Evaluation:** depends on the feedback mechanism

# Neighborhood models

Users	Items							
	1	0		1				
			0	0			1	
	0	1	0		1			
	1		0	0	1			

- **User-based model:**
  - Given a user  $u$ , find the most similar users  $\{u\}$
  - (Similar rows of the utility matrix)
  - Predict items  $v$  with high feedback by similar users, not yet consumed by  $u$

# Neighborhood models

		Items					
		1	0		1		
				0	0		1
		0	1	0		1	
Users			1	0	0	1	

- **User-based model:**
  - Given a user  $u$ , find the most similar users  $\{u\}$
  - (Similar rows of the utility matrix)
  - Predict items  $v$  with high feedback by similar users, not yet consumed by  $u$

- **Item-based model:**
  - Find items  $v'$  similar to those consumed by  $u$
  - (Similar columns of the utility matrix)
  - Predict those which have not yet been consumed by  $u$

# Neighborhood models: Issues

		1	0		1		
				0	0		1
0	1	0			1		
1			0		1		

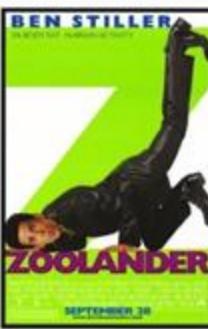
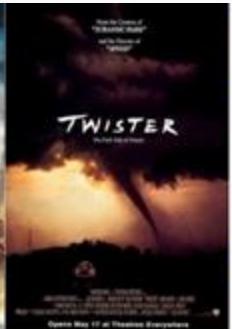
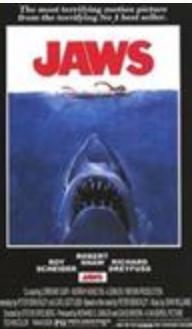
- Conceptually simple, but the devil is in the details!
  - How do you define **similarity** between users? Between items? By which metric?
  - How do you **aggregate feedback** over the neighborhood (from multiple similar users)?
  - User-based: Scaling issues (large user base, user turnover)
  - Item-based: Usually scales better and more stable, but might miss changes / trends
- Depending on feedback type, this can be difficult to scale with big data
  - Binary feedback  $\Rightarrow$  Jaccard similarity, **MinHash+LSH** will work
  - Otherwise ... ? Most spatial data structures are not robust to missing data!

In real life (big data) situations, data will be large (both rows and columns) and most data will be missing:  
We need to work **smarter!** → Latent factor models

- **Latent factor models**
- Flexible framework for feedback modeling
  - Objective can be **tuned to match feedback** mechanism (e.g., ★★★☆☆ vs play counts)
  - Secondary objectives can be added (item bias, regularization, etc.)
- Usually easy to **parallelize** and scale up training
  - E.g.: alternating least squares.
  - Users are independent (conditional on items), and vice versa
- Learned representation is **low-rank** and dense
  - Integrates well with spatial data structures
  - Rank parameter provides control on complexity  $\Leftrightarrow$  expressivity

## Ratings matrix

### Movie Ratings

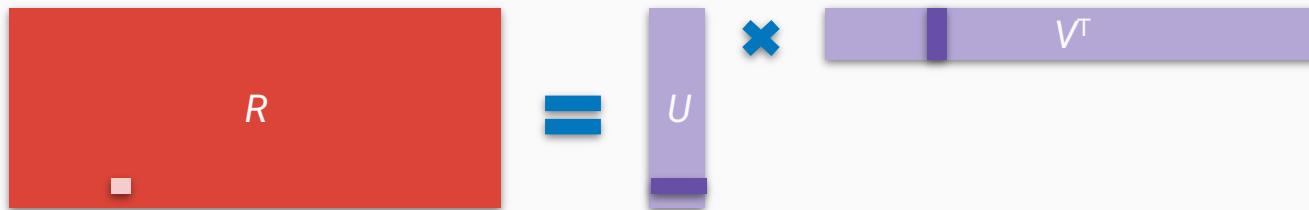


Alex	1	1	3	5	2
Brett	4.25	1.25	3	3.25	4.75
Casey	2	5	3	1	1
Drew	4.75	3.25	3	1.25	4.25
Emory	2.25	4.5	4.5	4.5	2.25
Frankie	2	1.75	2	1.75	2

# Latent factor models

- Bi-linear model of user-item interactions:

$$\min_{U,V} \sum_{(i,j) \in \Omega} (R_{ij} - \langle U_i, V_j \rangle)^2$$



- $U$  and  $V$  are embeddings of users and items into a common vector space
- Model parameters  $(U, V)$  are learned to predict sparse observations  $R$

# *Reconstructing ratings by matrix multiplication*

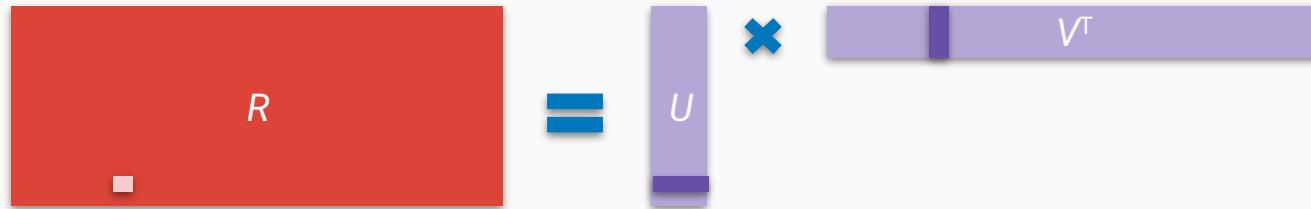
FP			
A	0	1	0
B	0.75	0.5	0
C	0	0	1
D	0.75	0	0.5
E	0	0.75	0.75
F	0.25	0.25	0.25

FC	H	J	S	T	Z
	5	1	2	1	5
	1	1	3	5	2
	2	5	3	1	1
Rat	H	J	S	T	Z
A	1	1	3	5	2
B	4.25	1.25	3	3.25	4.75
C	2	5	3	1	1
D	4.75	3.25	3	1.25	4.25
E	2.25	4.5	4.5	4.5	2.25
F	2	1.75	2	1.75	2

# Alternating least squares

- Repeat:
  - Fix user factors  $U$ , solve for item factors  $V$
  - Fix item factors  $V$ , solve for user factors  $U$

$$\min_{U,V} \sum_{(i,j) \in \Omega} (R_{ij} - \langle U_i, V_j \rangle)^2$$



- When user factors are fixed:
  - Each item vector  $V_i$  can be solved independently of the others  $V_j \leftarrow \text{parallelism!}$
  - Optimization problem is equivalent to ordinary least squares regression

# Summary: ALS via OLS

- **Problem:** Neighborhood models have problems with sparse feedback matrices (most data is missing), as well as with integration of feedback.
- **Solution:** We decompose the feedback matrix (e.g. a ratings matrix  $R$ ) into two smaller matrices,  $U$  (representing the user preferences for particular features) and  $V$  (representing feature content of the items).
- **Approach:** We initialize  $U$  and  $V$  with random values (or with values from a SVD), then using OLS to alternate between steps where we treat the entries in the  $V$  matrix as the parameters and the  $U$  matrix as the data and a step where we treat the entries in the  $U$  matrix as the parameters and the entries in the  $V$  matrix as the data.
- **OLS:** We do this with OLS (iteratively / alternatively).
- Until **convergence**: RMSE between the between the dot product of  $U$  and  $V^T$  and the actually existing ratings in the  $R$  matrix converges.

# ALS step by step

- We start with a feedback matrix  $\mathbf{R}$  (dimensionality  $n \times m$ ).
- We pick the hyperparameter  $k$  as the number of features to consider (or get this from SVD as the number of non-negligible singular values).
- We initialize a matrix  $\mathbf{U}$  as  $n \times k$  and a matrix  $\mathbf{V}$  as  $m \times k$ .

OLS normal equation:

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

- We now compute (per user  $i$  from 1 to  $n$  for all existing ratings  $r$ ):

$$\mathbf{U}_i = (\mathbf{V}^T \mathbf{V})^{-1} \mathbf{V}^T \mathbf{R}_r$$

- Then, we compute (per item  $j$  from 1 to  $m$  for all existing ratings  $r$ ):

$$\mathbf{V}_i = (\mathbf{U}^T \mathbf{U})^{-1} \mathbf{U}^T \mathbf{R}_r$$

- Then compute  $\text{RMSE}(\mathbf{R}_r, \mathbf{UV}^T)$ . If this term doesn't change or drops under some threshold, stop. Otherwise repeat the last 2 steps.

# More tricky: Modeling implicit feedback

- Count data (e.g. number of times listened to some song) is hard to predict
  - And we don't really care about that anyway!
- Instead, predict **binary interaction**, but use **counts to weight** terms!

$R_{ij}$ : Raw interaction count

$\alpha$ : hyperparameter  
(how much repeated interactions matter)

$$\min_{U,V} \sum_{(i,j) \in \Omega} c_{ij} (p_{ij} - \langle U_i, V_j \rangle)^2$$

$$p_{ij} = \begin{cases} 1 & R_{ij} > 0 \\ 0 & R_{ij} = 0 \end{cases}$$
$$c_{ij} = 1 + \alpha R_{ij}$$

User showed interest

User did not

$C_{ij}$  = Confidence

# Modeling implicit feedback for recommendations

- Count data (e.g. number of times listened to some song) is hard to predict
  - And we don't really care about that anyway!
- Instead, predict **binary interaction**, but use **counts to weight** terms!

$R_{ij}$ : Raw interaction count

$\alpha$ : hyperparameter  
(how much repeated interactions matter)

$$\min_{U,V} \sum_{(i,j) \in \Omega} c_{ij} (p_{ij} - \langle U_i, V_j \rangle)^2$$

$$p_{ij} = \begin{cases} 1 & R_{ij} > 0 \\ 0 & R_{ij} = 0 \end{cases}$$
$$c_{ij} = 1 + \alpha R_{ij}$$

User showed interest

User did not

$C_{ij}$  = Confidence

- You don't have to use R directly. There are other implementations:
  - Simply drop low counts
  - Compress large values  $R_{ij} \rightarrow \log(1 + R_{ij})$

[Hu, Koren, Volinsky 2008]

# Another practical issue of recommender systems with latent factor models: Handling new items

- Beware: This approach gives **no representation** to items with **no interactions**
- ⇒ A new item will never be recommended until it has representation!
- This is known as the **cold-start** problem
- Solutions typically involve
  - Active promotion / manual curation
  - Complementing this with content-based modeling

# Content-based models

Suppose you have observed features  $\mathbf{x}_j$  for each item

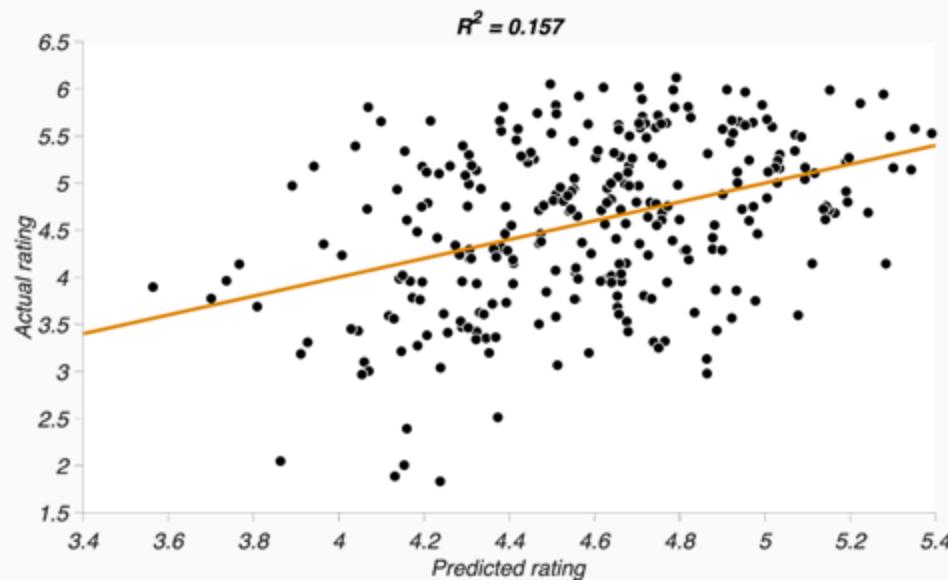
- News  $\Rightarrow$  topics, location, source
- Movies  $\Rightarrow$  genre, year, length, language, director, actors, ...
- Music  $\Rightarrow$  metadata + acoustic attributes

- **Content-based model**

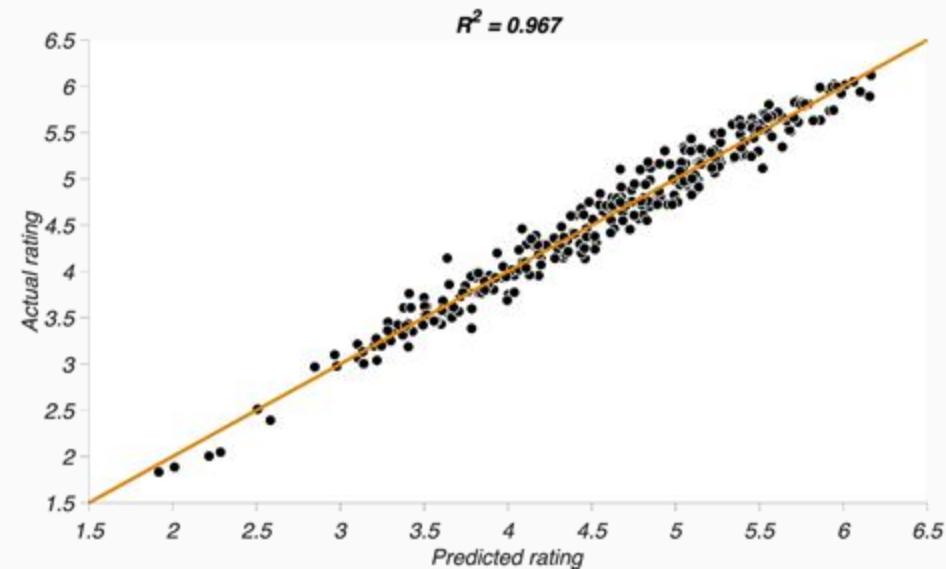
- Each user  $i$  gets their own interaction model  $\mathbf{u}_i$   
 $R_{ij} \approx \langle \mathbf{u}_i, \mathbf{x}_j \rangle$
- Like LF model, but the **item factors** are **explicit**
- Can be limiting / over-constrained

# Challenge: What if the content features are not that predictive?

Prediction from audio features



Prediction from evoked emotions



# Content-based models

Suppose you have observed features  $\mathbf{x}_j$  for each item

- News  $\Rightarrow$  topics, location, source
- Movies  $\Rightarrow$  genre, year, length, language, director, actors, ...
- Music  $\Rightarrow$  metadata + acoustic attributes

## • Content-based model

- Each user  $i$  gets their own interaction model  $\mathbf{u}_i$   
$$R_{ij} \approx \langle \mathbf{u}_i, \mathbf{x}_j \rangle$$
- Like LF model, but the **item factors** are **explicit**
- Can be limiting / over-constrained

## • Content cold-start

- Train a LF model as before, item  $j \rightarrow \mathbf{v}_j$
- Then regress item factors from features  
$$\mathbf{v}_j \approx \mathbf{f}(\mathbf{x}_j)$$
- A new item can map into embedding space by learned mapping  $\mathbf{f}(\mathbf{x}_k)$  (e.g. linear regression)

# User warm-start

- What happens when a new user enters a system?
- Typical systems request some **demographic data**, and ask for **examples of things you like**
- These data are used to position you in the collaborative filter



Use search to find stations. Look for the icon to collect them.

Want help finding something new?

[Browse Genres](#)

# Evaluation of recommender systems

- $R_{ij} \approx \langle u_i, v_j \rangle$
- $R_{ij} \approx b_j$
- $R_{qj} \approx \text{Jaccard}(q, v_j)$

## Task

- Predict a ranked list of items
  - Could be from collaborative filter...
  - Could be from popularity...
  - Could even be from a search query...
- What is the workflow for recommender systems modeling?
- How should we evaluate a model?

# Working with recommender data

- It's tempting to think of observations  $(u, v)$  as independent, but they aren't!
  - Different items for the same user  $\Rightarrow$  not independent
  - Different users for the same item  $\Rightarrow$  not independent
- What are we actually **predicting**, and what do we **value**?
  - Typically we care about satisfying a **user** – given the list of recommendations
  - This should influence our evaluation criteria (beyond RMSE) – unseen relevant?
- Most interfaces provide several recommendations at once
  - Evaluate the **collection of recommendations** per user
  - **Average across users** to estimate system performance

# Partitioning data for recommender systems

- We often need to partition training data for validation / parameter tuning
- It's tempting to randomly split interactions, but this can fail badly, as recommendations are not necessarily independent of users/items.
- The model needs at least some history for each user that it will predict upon, and sequence could matter
- **⇒ Partition each user's observations into train / val separately**

# Evaluating recommender systems

- Modeling objectives are usually just a proxy for our main goal
- Early RecSys work focused on **mean squared error** (MSE) of star ratings
  - $\text{loss}(\textcolor{teal}{u}_i) = \mathbf{E}_j[(\textcolor{red}{R}_{ij} - \langle \textcolor{teal}{u}_i, \textcolor{blue}{v}_j \rangle)^2]$
  - (Thanks, NetFlix)

# Evaluating recommender systems

- Modeling objectives are usually just a proxy for our main goal
- Early RecSys work focused on **mean squared error** (MSE) of star ratings
  - $\text{loss}(\mathbf{u}_i) = \mathbb{E}_j [ (R_{ij} - \langle \mathbf{u}_i, \mathbf{v}_j \rangle)^2 ]$
  - (Thanks, NetFlix)
- Instead, think about how recommendations are **delivered**
  - Ranked list? (NetFlix, Google search results, Amazon)
  - One at a time? (Pandora, streaming radio, YouTube autoplay)
- ***Evaluation should reflect user behavior!***

# Bipartite ranking evaluation

- **Idea:** rank items in order of decreasing estimated relevance for the user
  - Item 5, Item 17, Item 209, Item 444, Item 682, Item 999, Item 128, ...
- Using **held-out interactions**, determine which items were **relevant (+)** or **irrelevant (-)**:
  - $\{17, 444, 682, \dots\} \Rightarrow -, +, -, +, +, -, -, \dots$
- Score the sequence of **+** and **-**
  - Ideally, all **+** come before all **-**
  - All **+**'s assumed to be equally good, all **-**'s equally bad  $\Rightarrow$  “Bipartite ranking”
  - Particularly useful for implicit feedback (clicks, plays)

**Ranking evaluations:** Metrics to assess recommendation models that focus on the order in which items are presented, with the assumption that the order is relevant for the user experience

**Prediction:** ordered list of items, **reference data:** held-out interactions

- **AUC** (area under ROC curve)
  - How often does a **+ interaction** rank ahead of a **- interaction**?
  - **-+---**  $\Rightarrow (3 + 2 + 2) / (3 * 4) = 7/12 = 0.583$

**Ranking evaluations:** Metrics to assess recommendation models that focus on the order in which items are presented, with the assumption that the order is relevant for the user experience

**Prediction:** ordered list of items, **reference data:** held-out interactions

- **AUC (area under ROC curve)**
  - How often does a **+ interaction** rank ahead of a **- interaction**?
  - **-+---**  $\Rightarrow (3 + 2 + 2) / (3 * 4) = 7/12 = 0.583$
  
- **Average Precision (AP)**
  - For each **+ interaction**, what fraction of higher-ranked items were also **positive**?
  - **-+---**  $\Rightarrow \frac{1}{3} * (\frac{1}{2} + \frac{1}{2} + \frac{3}{5}) = 0.533$  [the lead  $1/3 = 1/n$  yields the average]

**Ranking evaluations:** Metrics to assess recommendation models that focus on the order in which items are presented, with the assumption that the order is relevant for the user experience

**Prediction:** ordered list of items, **reference data:** held-out interactions

- **AUC** (area under ROC curve)
  - How often does a **+ interaction** rank ahead of a **- interaction**?
  - **-+---**  $\Rightarrow (3 + 2 + 2) / (3 * 4) = 7/12 = 0.583$
- **Average Precision (AP)**
  - For each **+ interaction**, what fraction of higher-ranked items were also **positive**?
  - **-+---**  $\Rightarrow \frac{1}{3} * (\frac{1}{2} + \frac{1}{2} + \frac{3}{5}) = 0.533$  [the lead  $1/3 = 1/n$  yields the average]
- **Reciprocal rank (MRR)**
  - The inverse rank position of first **+ interaction**
  - **-+---**  $\Rightarrow \frac{1}{2}$

**Ranking evaluations:** Metrics to assess recommendation models that focus on the order in which items are presented, with the assumption that the order is relevant for the user experience

**Prediction:** ordered list of items, **reference data:** held-out interactions

- **AUC (area under ROC curve)**
  - How often does a **+ interaction** rank ahead of a **- interaction**?
  - **-+---**  $\Rightarrow (3 + 2 + 2) / (3 * 4) = 7/12 = 0.583$
- **Average Precision (AP)**
  - For each **+ interaction**, what fraction of higher-ranked items were also **positive**?
  - **-+---**  $\Rightarrow \frac{1}{3} * (\frac{1}{2} + \frac{1}{2} + \frac{3}{5}) = 0.533$  [the lead  $1/3 = 1/n$  yields the average]
- **Reciprocal rank (MRR)**
  - The inverse rank position of first **+ interaction**
  - **-+---**  $\Rightarrow \frac{1}{2}$

Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR): Users are assumed to be independent, and scores are averaged across users at evaluation time.

# Beyond bipartite ranking metrics

Bipartite ranking metrics work well when the feedback *\*is\** binary (clicks, likes, views, listens)

There is a space for a ranking evaluation metric (result is a ranked list) that takes graded relevance (e.g. star ratings) into account.

Most commonly used in this scenario:

Normalized Discounted Cumulative Gain (**NDCG**)

NDCG allows to model graded relevance, while also considering the ranking of the results.

Philosophy: "Reward the model for ranking good items higher, especially the best ones."

# How NDCG works (mathematically)

Movie	Stars
Avatar (2009)	2
Big Short (2015)	5
Cats (2019)	1
Dune (2021)	4
Elysium (2013)	3

Key idea: There is an ideal order – NDCG represents how well the ranking our model produces compares to that ideal (with a metric ranging from 0 to 1)

$$NDCG@k = \frac{DCG@k}{IDCG@k} \quad CG@k = \sum_{i=1}^k rel_i$$

Ideal order: B, D, E, A, C      CG@3 = 12

How does a given recommended order compare to that? D,C,B,E,A    CG@3 = 10

In addition to the sum, the presentation order should matter too: It is better to serve higher value items earlier. Items later in the list should matter less.

$$DCG@k = \sum_{i=1}^k \frac{rel_i}{\log_2(i + 1)}$$

This is the linear case. If you want to reward bringing high-relevance items more, use exponential gain:

$$DCG@k = \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(i + 1)}$$

# In real life, things are even more complicated...

- In practice, recommender systems exist in a **feedback loop**
- These are extremely difficult to measure offline from **observational data**
  - Interpret ranking metrics with healthy skepticism!
- Competing models are often evaluated by A/B testing, and measuring some dependent variable
  - Engagement, sales, etc.
- Recently, focus is shifting to reinforcement learning and causal models

# Evaluation summary

- Properly evaluating a recommender system is not easy!
- Ranking metrics are a start, but there's much more to it than “accuracy”
  - Diversity? Novelty? Serendipity?
  - Efficiency? Ease of use?
  - Explainability / transparency?
  - Adverse effects on users?

# Next week

- Graph algorithms (esp. page rank)
- Spatial search (vector databases)

# After that

- Ethics (filter bubbles, polarization)
- Privacy

**Q & R**



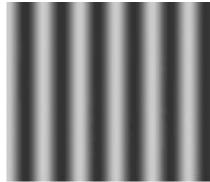
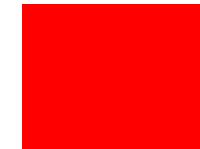
Smallest font



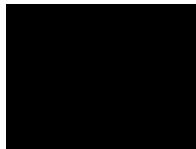
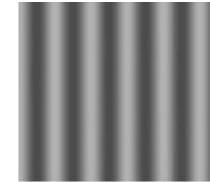
Please turn off and put  
away your cell phone



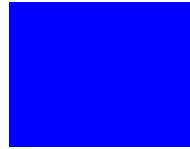
# Calibration slide



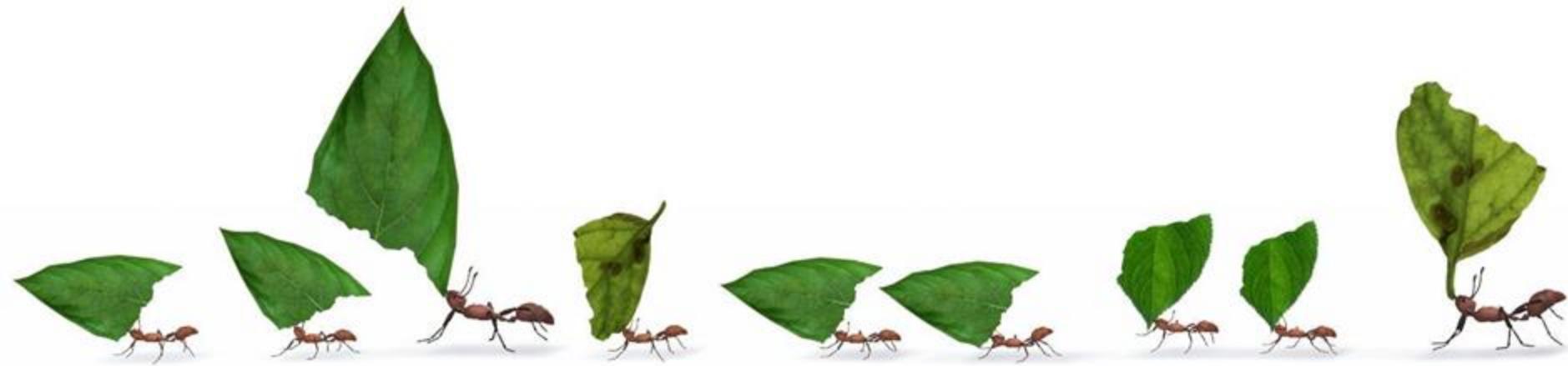
These slides are meant  
to help with note-taking  
They are no substitute  
for lecture attendance



Smallest font



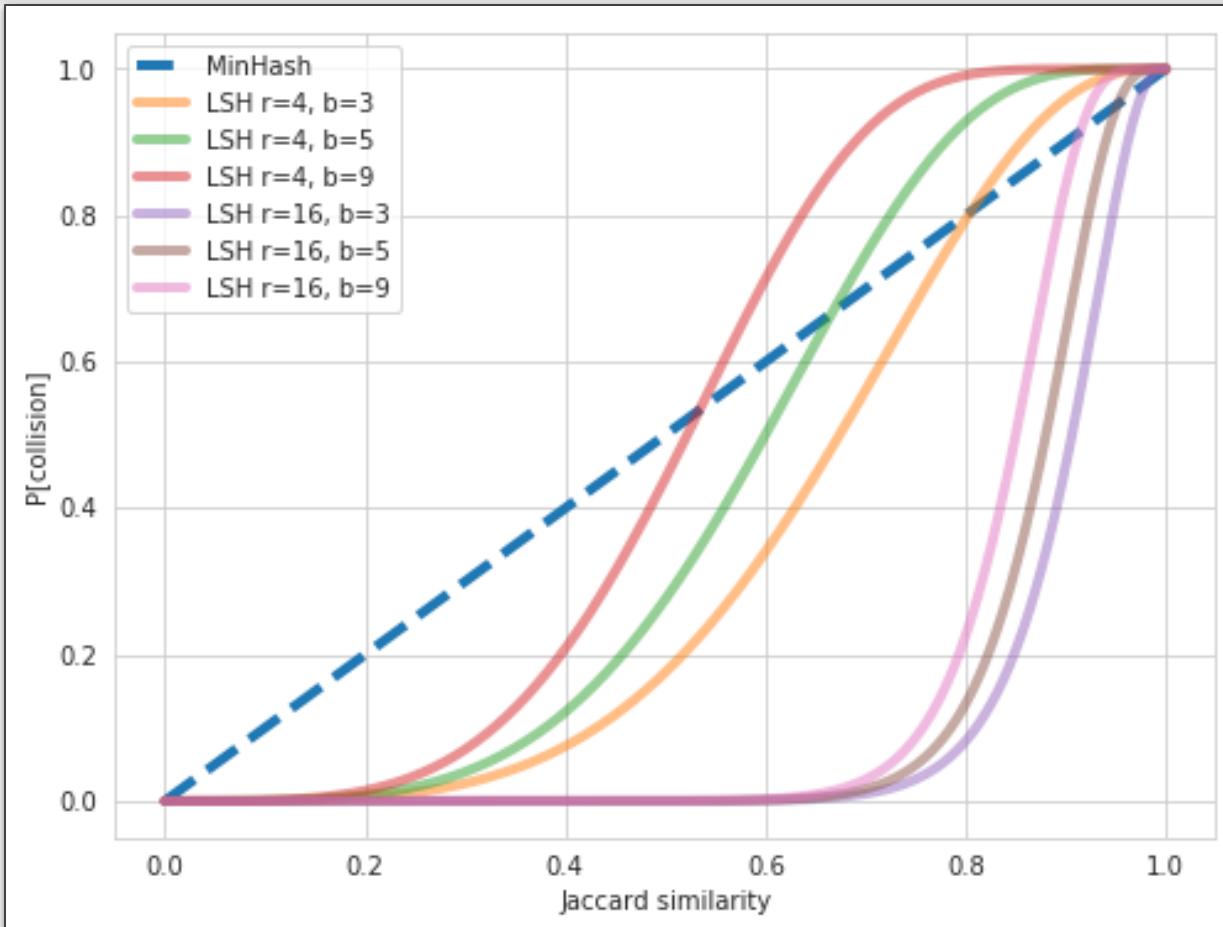
# Big Data



# Beyond set similarity: Spatial similarity search Graph algorithms

DS-GA 1004: Big Data

# CDS: Locality-sensitive hashing (LSH)



# Orientation

Search

Relevance from **content**  
similarity between query  
and document

**Personalization** of  
relevance from  
explicit or implicit  
feedback

Recommendation

Relevance from  
network **structure**

Graph  
algorithms

Beyond simple sets:

1) Bags (multi-sets)

What if you don't just want to consider whether an item appears (in a set), but also how **\*often\*** it does so (in a multi-set/bag):

## Ruzicka similarity

[Chen, Philbin, Zisserman 2008]

- Idea: reduce bags to sets by uniquely identifying each repetition  $\min(x_1, y_1) = \min(1, 2) = 1$

$\{\text{dog}\}$	$\rightarrow \{\text{dog}_1\}$	$\mathbf{A} \quad \mathbf{B} \quad \mathbf{C} \quad \mathbf{D}$	$\max(x_1, y_1) = \max(1, 2) = 2$
$\{\text{dog}, \text{dog}\}$	$\rightarrow \{\text{dog}_1, \text{dog}_2\}$	$X = [1, 3, 5, 7]$	$\min(x_2, y_2) = \min(3, 3) = 3$
$\{\text{dog}, \text{dog}, \text{dog}\}$	$\rightarrow \{\text{dog}_1, \text{dog}_2, \text{dog}_3\}$	$Y = [2, 3, 1, 6]$	$\max(x_2, y_2) = \max(3, 3) = 3$

- Jaccard on expanded sets = **Ruzicka similarity** on original bags  $\min(x_3, y_3) = \min(5, 1) = 1$

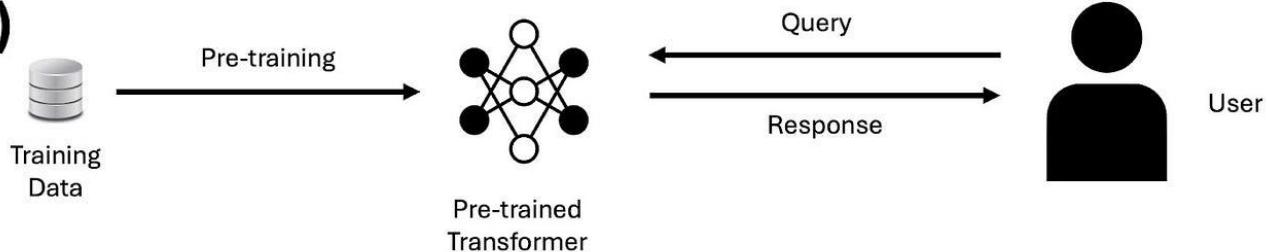
$$R(\mathbf{A}, \mathbf{B}) = \frac{\sum_i \min(\mathbf{A}[i], \mathbf{B}[i])}{\sum_j \max(\mathbf{A}[j], \mathbf{B}[j])}$$

$\Sigma(\min(x_i, y_i)) = 1 + 3 + 1 + 6 = 11$        $\max(x_3, y_3) = \max(5, 1) = 5$   
 $\Sigma(\max(x_i, y_i)) = 2 + 3 + 5 + 7 = 17$        $\min(x_4, y_4) = \min(7, 6) = 6$   
 $R(X, Y) = 11 / 17 \approx 0.65$        $\max(x_4, y_4) = \max(7, 6) = 7$

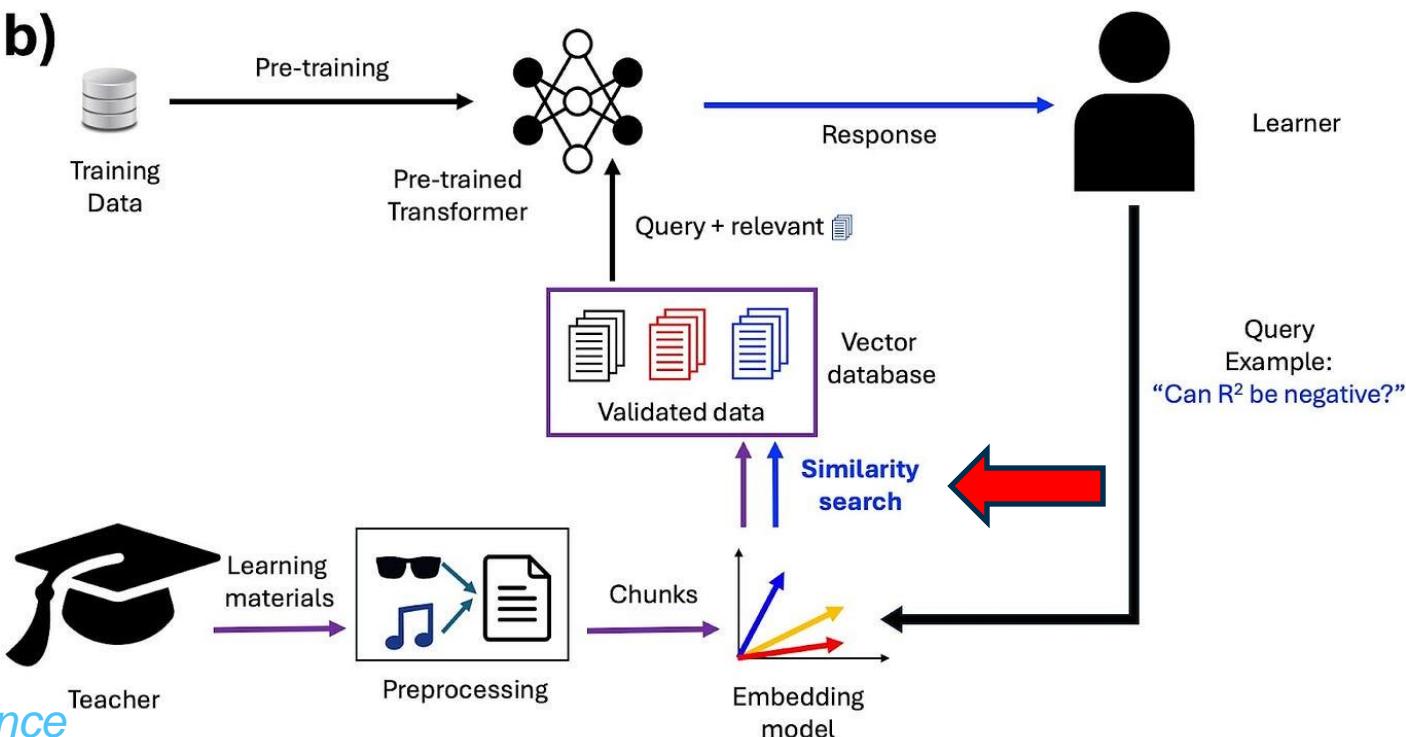
Beyond simple sets:

2) Spatial similarity search  
(as in a vector database)

# A typical use case a)



b)



# Cosine similarity is often a suitable metric to compare documents (and other high-dimensional objects)

- For instance, one could express documents in terms of the magnitude of their embedding dimensions.
- Cosine similarity conceptualizes the similarity of two vectors **a** and **b** in terms of the angle  $\theta$  between them, regardless of their length.
- This makes intuitive sense:

$$\text{cosine similarity} = \cos(\theta) = \frac{\mathbf{a}^T \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \frac{\mathbf{a}^T \mathbf{b}}{\sqrt{\mathbf{a}^T \mathbf{a}} \sqrt{\mathbf{b}^T \mathbf{b}}}$$

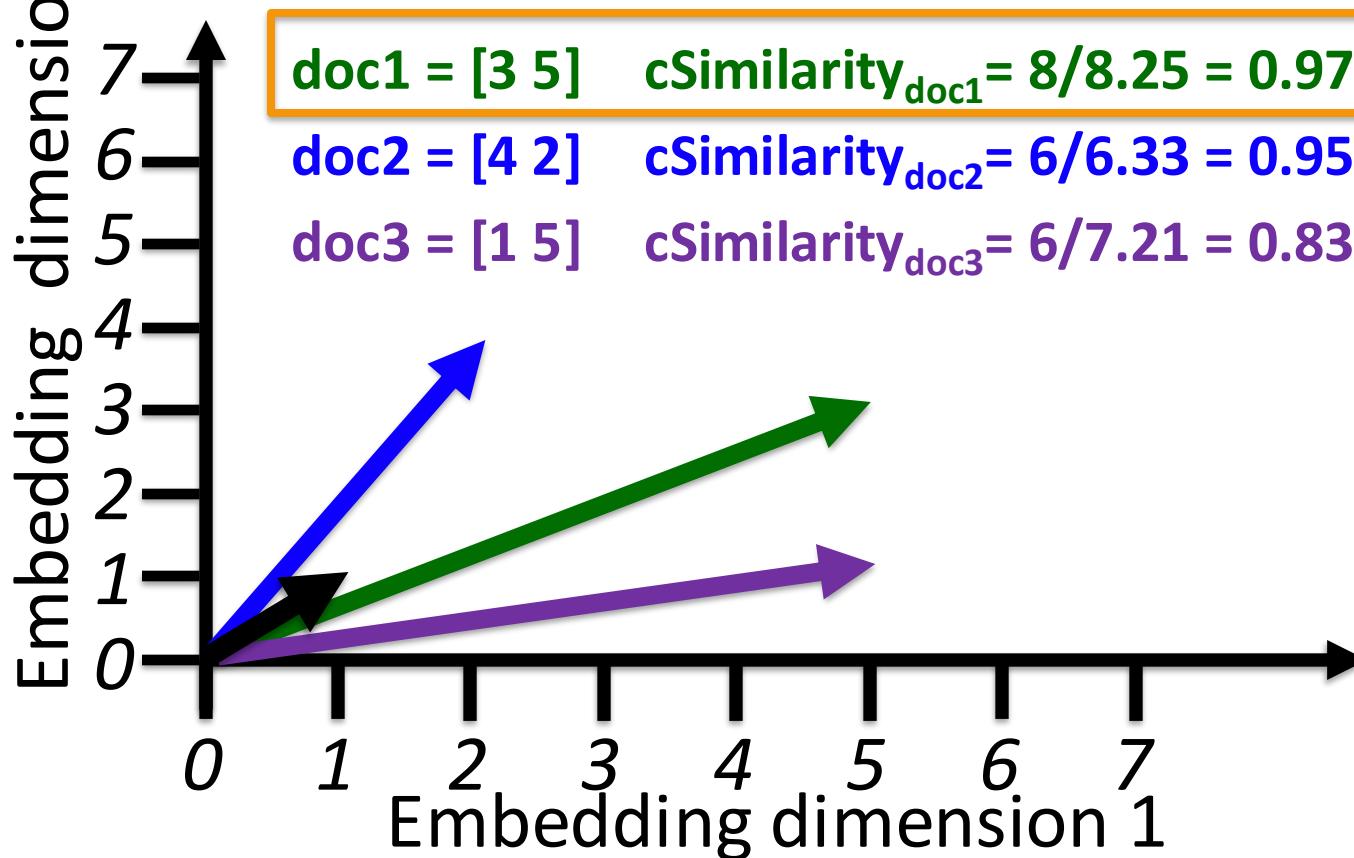
$$\cos(0) = 1$$

$$\cos(90) = 0$$

$$\cos(180) = -1$$

# Determining relevance with cosine similarity

$\text{doc}_0 = [1 \ 1]$



# Locality Sensitive Hashing (LSH) for cosine similarity

- Computing cosine similarity is straightforward between any two vectors, but can be computationally expensive if the dimensionality  $d$  of the vectors is large.
- $N$  will be very large (many billions of vectors or more) in a modern vector database, so time complexity is  $O(Nd)$ .
- The idea is then to use LSH to whittle down all vectors to a candidate set.
- Full cosine similarity is only computed for the vectors in the candidate set.
- LSH is valid but a bit abstract for sets. It is more geometrically intuitive here.
- Charikar (2002) introduced LSH for cosine similarity by grouping similar items into the same “bucket”, by dividing the vector space in half.
- This data partitioning works by using random hyperplanes to slice the vector space.
- The hyperplane divides the space into two halves. The hash depends on which side it falls on. If they fall on the same side, the vectors “collide”.
- The probability of a collision is related to the (cosine) similarity of the vectors.

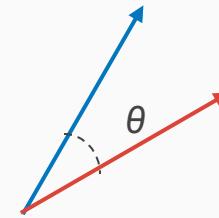
# LSH for cosine similarity

[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

Straightforward, but can we approximate the cosine similarity between these vectors without computing the cosine similarity between them?



Idea: Pick a random direction ( $w$ ) in the space. Technically, we do this by choosing  $w$  uniformly at random from a sphere in  $d$ -dimensional space.

# LSH for cosine similarity

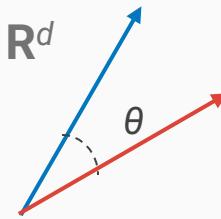
[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  randomly but uniformly from the unit sphere in  $\mathbb{R}^d$

- $h_w(x) = 1$  if  $w^T x \geq 0$   
 $= -1$  if  $w^T x < 0$



# LSH for cosine similarity

[Charikar 2002]

- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

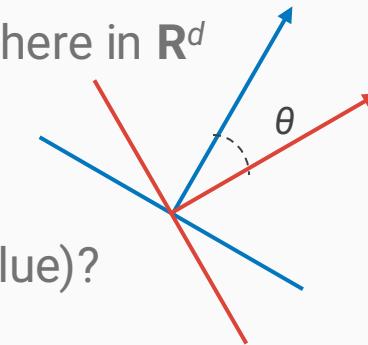
$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  randomly but uniformly from the unit sphere in  $\mathbb{R}^d$

- $h_w(x) = 1$  if  $w^T x \geq 0$   
 $= -1$  if  $w^T x < 0$

- What's the probability of a collision (both same hash value)?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)]$



Two vectors ( $u, v$ ) separated by an angle  $\theta$  define a geometric region between them. The randomly chosen hyperplane ( $w$ ) partitions space. The probability that the hyperplane falls between the two vectors (shaded region) directly relates to the angle  $\theta$ :

# LSH for cosine similarity

[Charikar 2002]

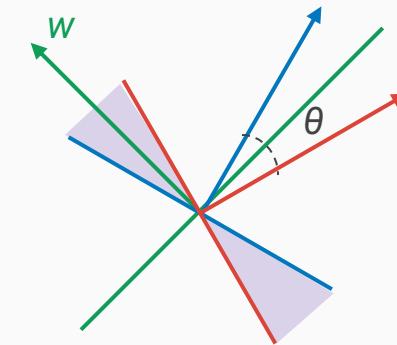
- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  uniformly from the sphere in  $\mathbb{R}^d$ 
  - $h_w(x) = 1$  if  $w^T x \geq 0$
  - $= -1$  if  $w^T x < 0$

- What's the probability of collision?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)] = 1 - P[w \text{ in shaded region}]$



# LSH for cosine similarity

[Charikar 2002]

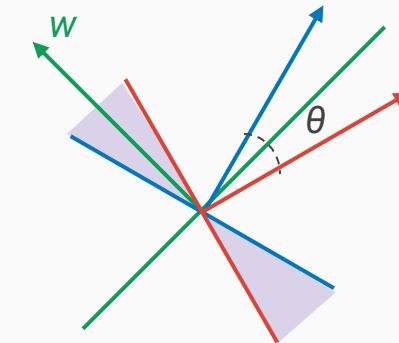
- What if we want to compare vectors  $u, v \in \mathbb{R}^d$  by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector  $w$  uniformly from the sphere in  $\mathbb{R}^d$ 
  - $h_w(x) = 1$  if  $w^T x \geq 0$
  - $= -1$  if  $w^T x < 0$

- What's the probability of collision?

- $P[h_w(u) = h_w(v)] = 1 - P[h_w(u) \neq h_w(v)]$   
 $= 1 - P[w \text{ in shaded region}]$   
 $= 1 - 2 \cdot |\theta| / 2\pi$   
 $= 1 - |\theta|/\pi$



Not exactly  $\cos(\theta)$ , but monotonically decreasing with  $|\theta| \Rightarrow$  same rank-ordering

In Charikar's method, the signal amplification for LSH does not come from rows and blocks, but from **multiple projections**:

- $P[\text{No collision} \mid \text{single projection}] = \theta/\pi$

A single hyperplane hash is very coarse (the probability of two vectors colliding by getting the same hash value is high), which leads to many false positives (akin to having many stop words in sets)

Projections  $m$ : How many ( $m$ ) random hyperplanes are used to partition the vector space

- $P[\text{All projections do not collide} \mid m \text{ projections}] = (\theta/\pi)^m$
- $P[\text{At least one Collision} \mid m \text{ projections}] = 1 - (\theta/\pi)^m$

So much for similarity search  
Both with sets and beyond  
Now: Relevance from graphs

# PageRank



# Early web search (pre Google)

- Early search engines relied on matching text in **query** to text in **web page**
- Pages were crawled (by “spiders”) at regular intervals and added to an index
- We’ve already discussed some tools for indexing and searching
- ***What could/did go wrong by relying on this approach?***



# Spam attacks



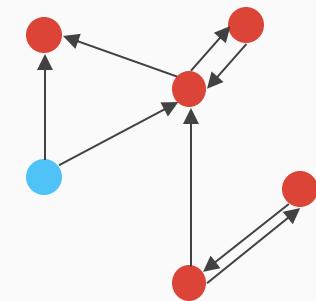
- Imagine that you want to get lots of traffic to your website
- You know that it is indexed regularly by search engines
- **Idea:** Discreetly fill your page with \*all\* of the **most popular search terms**
- End result: **\$\$\$** selling items irrelevant to the user query

# PageRank: use the structure of the network itself!

- Key insight: the **structure** of the network contains information!
- Publishers are more likely to link to pages that they trust
- It's **easy** to make a spammy page
- It's **hard** to get other people to link to it

# The random surfer model

- Imagine the web as a directed graph
  - Nodes = pages
  - Directed Edges = links (have a direction)
- Model a user's activity as a random walk
  - $P[\text{Going to page } v \mid \text{Currently at page } u] = [u \rightarrow v] / \text{out-degree}(u)$   
[0 or 1] total number of edges
- Users are more likely to land at pages with **high in-degree**
  - What is the steady-state distribution  $P[v]$ ?



# Markov chains

- Let  $M[v, u] = P[v | u]$ 
  - Columns ( $u$ ) = current states (N pages)                          Rows ( $v$ ) = next states (N pages)
- $M$  is a **stochastic matrix**: non-negative, each column sums to 1

# Markov chains

- Let  $\mathbf{M}[v, u] = P[v | u]$ 
  - Columns ( $u$ ) = current states (N pages) Rows ( $v$ ) = next states (N pages)
- $\mathbf{M}$  is a **stochastic matrix**: non-negative, each column sums to 1
- Let  $\mathbf{p}$  be a non-negative vector in  $\mathbb{R}^N$  that sums to 1
  - $(\mathbf{Mp})[v]$  marginalizes over  $u$  to compute probability of state  $v$

# Markov chains

- Let  $\mathbf{M}[v, u] = P[v | u]$ 
  - Columns ( $u$ ) = current states (N pages) Rows ( $v$ ) = next states (N pages)
- $\mathbf{M}$  is a **stochastic matrix**: non-negative, each column sums to 1
- Let  $\mathbf{p}$  be a non-negative vector in  $\mathbb{R}^N$  that sums to 1
  - $(\mathbf{Mp})[v]$  marginalizes over  $u$  to compute probability of state  $v$
- So: One step maps

$$p[v] \rightarrow (\mathbf{Mp})[v] = \sum_u P[v | u] * p[u] = p'[v]$$

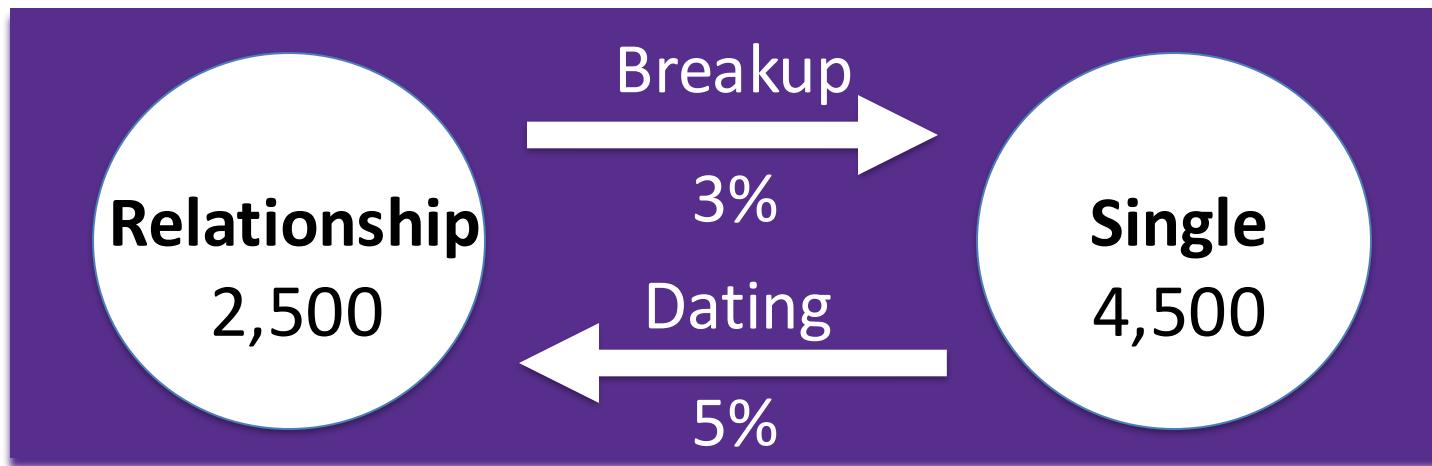
# Steady-state distributions

- A steady-state distribution satisfies the identity  $\mathbf{p} = \mathbf{M}\mathbf{p}$
- Such a  $\mathbf{p}$  exists if there is a *path* connecting every pair  $u \rightarrow v$  in a finite number of steps
  - If that is the case, we say that  $\mathbf{M}$  is **irreducible** or **strongly connected** [more on this soon]

# Steady-state distributions

- A steady-state distribution satisfies the identity  $\mathbf{p} = \mathbf{M}\mathbf{p}$
- Such a  $\mathbf{p}$  exists if there is a *path* connecting every pair  $u \rightarrow v$  in a finite number of steps
  - If that is the case, we say that  $\mathbf{M}$  is *irreducible* or *strongly connected* [more on this soon]
- $\mathbf{p}$  = **eigenvector** of  $\mathbf{M}$  with **eigenvalue 1**
  - The largest possible for a stochastic matrix!
- $\text{PageRank}(u) = p[u] = \text{probability of random surfer being at node } u$
- *This is all a bit abstract, let's give a vivid, real life example of such modeling.*

Spoiler: The Eigenvector of a Markov matrix represent the long term equilibrium of a dynamic system:



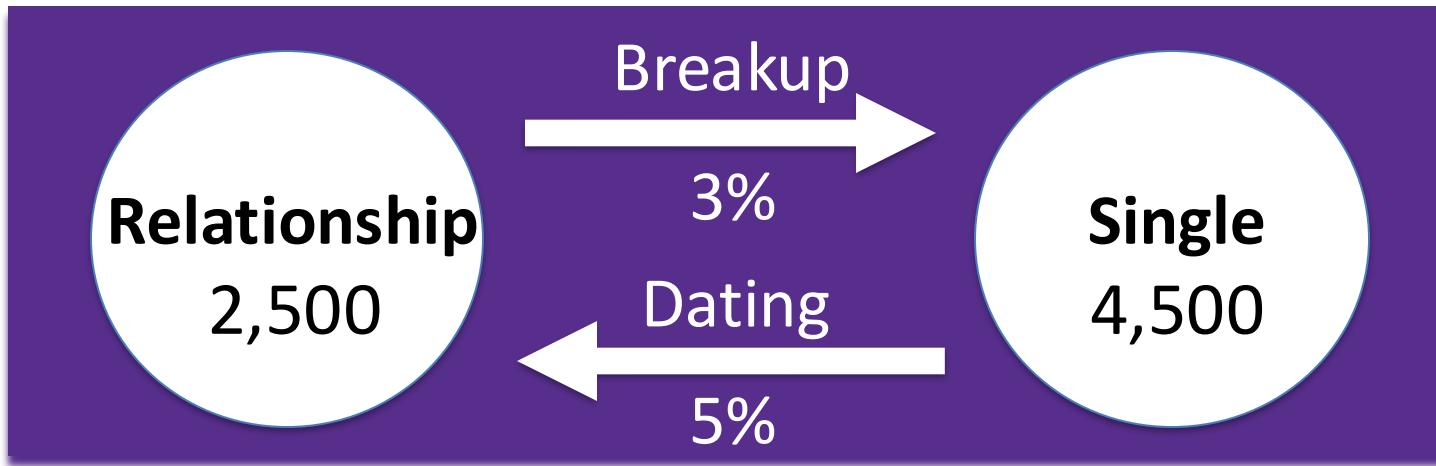
After the 1<sup>st</sup> month:

$$\text{In relationships: } 0.97(2,500) + 0.05(4,500) = 2,650$$

$$\text{Single: } 0.03(2,500) + 0.95(4,500) = 4,350$$

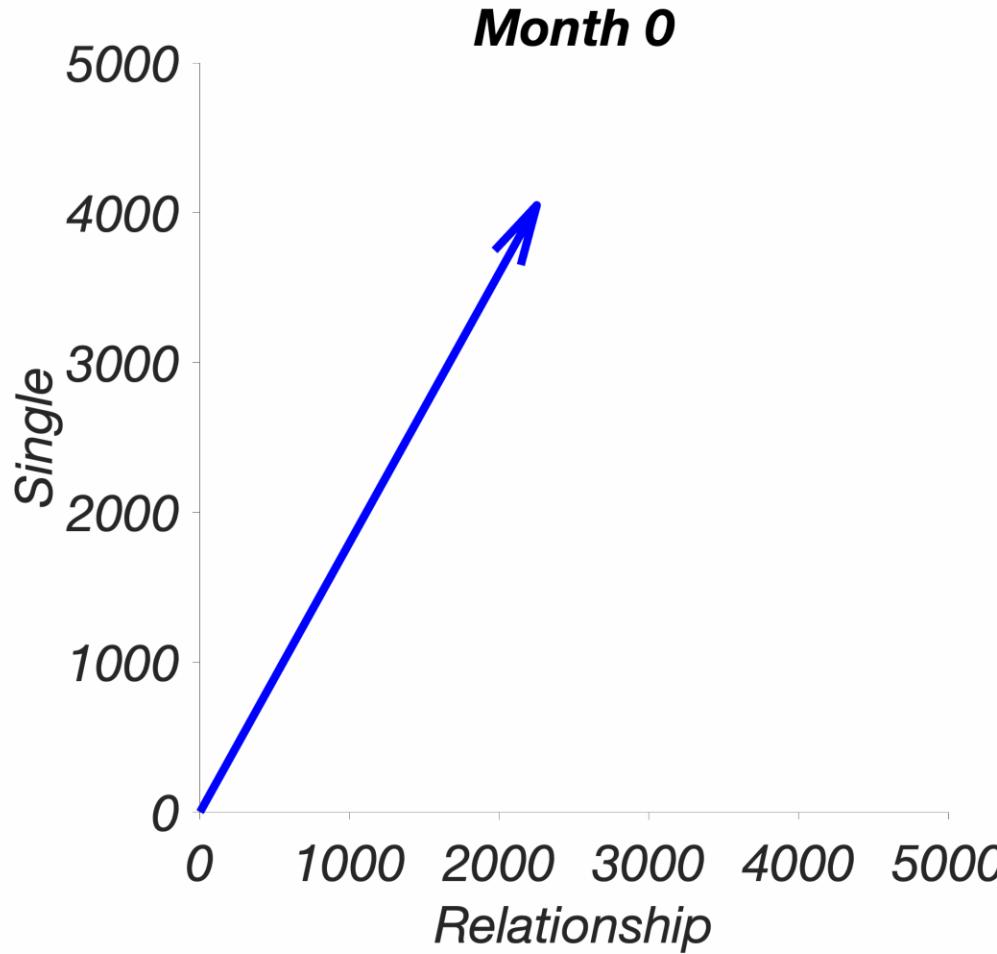
How about the long term?

# Modeling this with a Markov Matrix

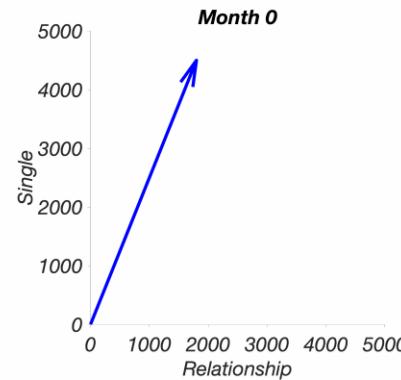
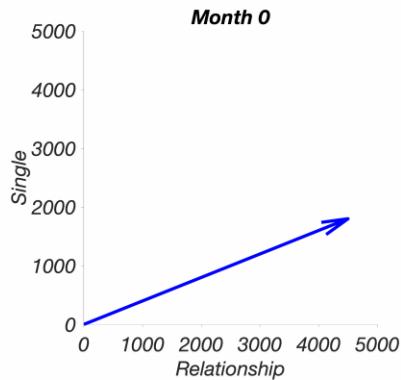
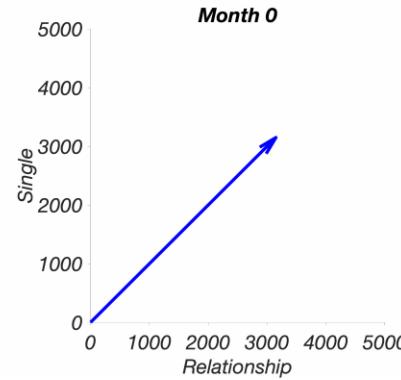
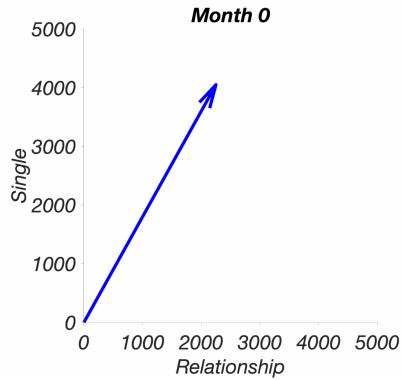


$$\begin{pmatrix} 0.97 & 0.05 \\ 0.03 & 0.95 \end{pmatrix} \begin{pmatrix} R_0 \\ S_0 \end{pmatrix} = \begin{pmatrix} R_{+1} \\ S_{+1} \end{pmatrix}$$

# (Where) does this system stabilize?



# Regardless of starting position

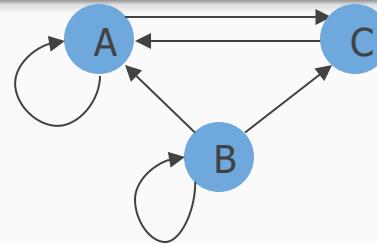


It stabilizes on the dominant Eigenvector. Eigenvalue: 1

# Markov chain Eigenvectors

- This works fine (if you have small data). But:
- Standard eigenvector solvers do not scale to the web (or big data in general)
  - $O(N^3)$  cost to solve in general, can be smaller if sparse, but  $N$  can still be huge!
- Instead, use **power iteration** (aka the “power method”)
  - Initialize  $\mathbf{p}_0[u] \leftarrow 1/N$  (uniform distribution)
  - For  $i = 1, 2, \dots, T_{\max}$ 
    - $\mathbf{p}_i \leftarrow \mathbf{M} \mathbf{p}_{i-1}$  ( $\mathbf{p}_i = \mathbf{M}^i \mathbf{p}_0$ ) [This can be parallelized over rows of  $\mathbf{M}$ ]
- This will eventually converge to the stationary distribution
  - (If such a distribution exists...)

# Small network example of power iteration: Initialization



**M**

$p_1$
?
?
?

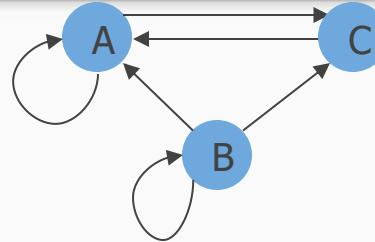


	A	B	C
A	$\frac{1}{2}$	$\frac{1}{3}$	1
B	0	$\frac{1}{3}$	0
C	$\frac{1}{2}$	$\frac{1}{3}$	0

$p_0$
$\frac{1}{3}$
$\frac{1}{3}$
$\frac{1}{3}$

- Initialize  $p_0[u] \leftarrow 1/N$
- For  $i = 1, 2, \dots, T_{\max}$ 
  - $p_i \leftarrow M p_{i-1}$

# Small network example of power iteration: $p_1$



**M**

$p_1$
11/18
2/18
5/18

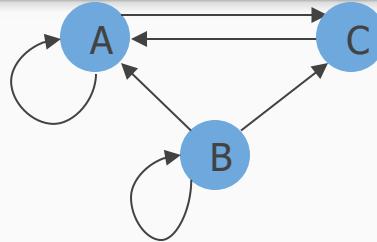


	A	B	C
A	$\frac{1}{2}$	$\frac{1}{3}$	1
B	0	$\frac{1}{3}$	0
C	$\frac{1}{2}$	$\frac{1}{3}$	0

$p_0$
$\frac{1}{3}$
$\frac{1}{3}$
$\frac{1}{3}$

- Initialize  $p_0[u] \leftarrow 1/N$
- For  $i = 1, 2, \dots, T_{\max}$ 
  - $p_i \leftarrow M p_{i-1}$

# Small network example of power iteration: $p_2$



**M**

$p_2$
67/108
4/108
37/108

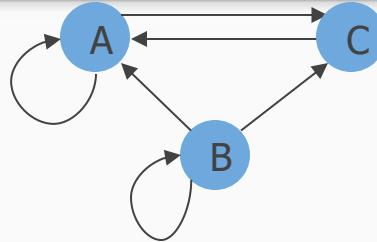


	A	B	C
A	$\frac{1}{2}$	$\frac{1}{3}$	1
B	0	$\frac{1}{3}$	0
C	$\frac{1}{2}$	$\frac{1}{3}$	0

$p_1$
11/18
2/18
5/18

- Initialize  $p_0[u] \leftarrow 1/N$
- For  $i = 1, 2, \dots, T_{\max}$ 
  - $p_i \leftarrow M p_{i-1}$

# Small network example of power iteration: $p_3$



M

$p_3$
431/648
8/648
209/648



	A	B	C
A	$\frac{1}{2}$	$\frac{1}{3}$	1
B	0	$\frac{1}{3}$	0
C	$\frac{1}{2}$	$\frac{1}{3}$	0

$p_2$
67/108
4/108
37/108

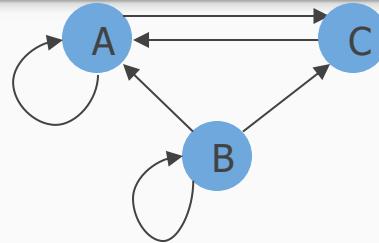
- Initialize  $p_0[u] \leftarrow 1/N$
- For  $i = 1, 2, \dots, T_{\max}$ 
  - $p_i \leftarrow M p_{i-1}$

# Small network example of power iteration: Steady state

The pagerank vector



$p_n$
2/3
0
1/3



$M$

	A	B	C
A	$\frac{1}{2}$	$\frac{1}{3}$	1
B	0	$\frac{1}{3}$	0
C	$\frac{1}{2}$	$\frac{1}{3}$	0

$p_n$
2/3
0
1/3

- Initialize  $p_0[u] \leftarrow 1/N$
- For  $i = 1, 2, \dots, T_{\max}$ 
  - $p_i \leftarrow M p_{i-1}$

# If you want to check your work in Python

- $M[v, u] = P[v | u]$  (probability of going to v from u)
  - Each column is a probability distribution
  - $\Rightarrow M.sum(axis=0)$  should be all ones
- `evals, evecs = np.linalg.eig(M)` is almost, but not quite, what we want

# If you want to check your work in Python

- $M[v, u] = P[v | u]$  (probability of going to v from u)
  - Each column is a probability distribution
  - $\Rightarrow M.sum(axis=0)$  should be all ones
- `evals, evecs = np.linalg.eig(M)` is almost, but not quite, what we want
- Remember:
  - `evecs` have unit Euclidean ( $L_2$ ) norm:  $v = v / \|v\|_2$
  - Probability distributions have unit  $L_1$  norm:  $p = v / \|v\|_1 \neq v$

# If you want to check your work in Python

- $M[v, u] = P[v | u]$  (probability of going to  $v$  from  $u$ )
  - Each column is a probability distribution
  - $\Rightarrow M.sum(axis=0)$  should be all ones
- `evals, evecs = np.linalg.eig(M)` is almost, but not quite, what we want
- Remember:
  - $evecs$  have unit Euclidean ( $L_2$ ) norm:
  - Probability distributions have unit  $L_1$  norm:
- So you have to re-normalize each column:
  - `evecs /= np.abs(evecs).sum(axis=0, keepdims=True)`

$$\mathbf{v} = \mathbf{v} / \|\mathbf{v}\|_2$$

$$\mathbf{p} = \mathbf{v} / \|\mathbf{v}\|_1 \neq \mathbf{v}$$

You may also have to flip the sign of  $v$ .

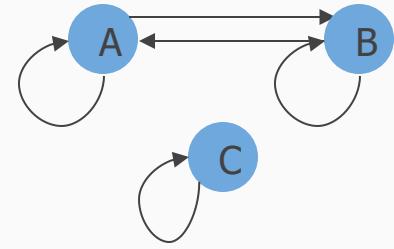
If  $v$  is an eigenvector, so is  $-v$ !

# Requirement:

## The graph must be connected!

- If the graph is not connected, there is not a **unique leading eigenvector (stationary distribution)**
- The **requirements** for PageRank **are not met**.

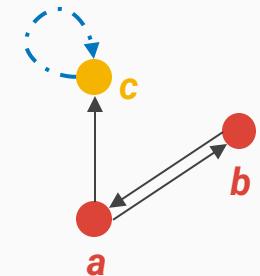
Remember: we need every vertex to be reachable by every other vertex!



	A	B	C
A	$\frac{1}{2}$	$\frac{1}{2}$	0
B	$\frac{1}{2}$	$\frac{1}{2}$	0
C	0	0	1

# What about “sinks”?

- A sink is a **vertex with no outgoing edges**, its column in M is all zeros.
  - This will not yield a valid probability distribution!
- Common fix: add a **self-loop** to any nodes having out-degree 0
  - **Nodes with outgoing edges** do not need to be modified
- Result is a well-formed transition matrix



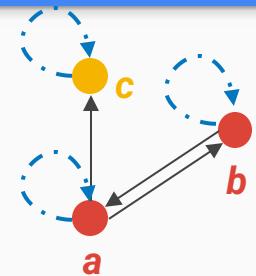
# “Spider traps”

- Node **c** is a **spider trap** in this graph – no outgoing links
- A random surfer landing at **c** can never leave!

	a	b	c	$p_0$
a	$\frac{1}{3}$	$\frac{1}{2}$	0	$\frac{1}{3}$
b	$\frac{1}{3}$	$\frac{1}{2}$	0	$\frac{1}{3}$
c	$\frac{1}{3}$	0	1	$\frac{1}{3}$

Power iteration

<b>p</b>
0
0
1

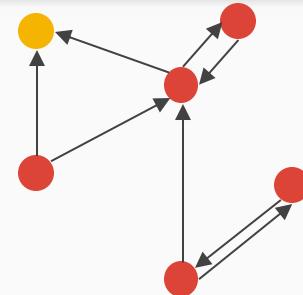


Traps are easy to create!

All it takes is one link from a well-connected vertex to cause serious damage!

# Solution: Teleportation / random restart

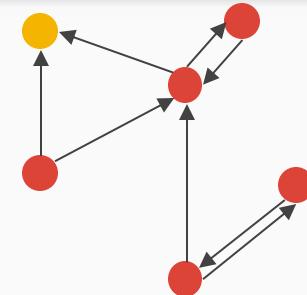
- The web is decidedly **not strongly connected**
- Not all pages have outward links (column sum = 0)



# Teleportation / random restart

- The web is decidedly **not strongly** connected
- Not all pages have outward links (column sum = 0)
- Solution: **teleportation!**
- With probability **a**, follow the links  
With probability **1-a**, jump uniformly at random

$$M[v, u] \rightarrow a * M[v, u] + (1 - a) * 1/N$$



# Using PageRank to improve search

- PageRank uses network topology to score relevance of each vertex (page)
- $p[v] > p[u] \Rightarrow v$  is “better connected” than  $u$ 
  - It does not use content! But still a reasonable measure of importance
- Basic search implementation:
  - Use **text search** (LSH, etc.) to find candidate items
  - Use **pagerank** (and other cues) to order results
- Can we do better?

# Improving further: Personalizing PageRank

- The uniform teleportation model isn't realistic
  - Jumping to **any** page? Really?
- If  $e = [1, 1, 1, \dots, 1]$ , then PageRank + teleportation computes
$$p = (a * M + (1-a) * \frac{1}{N} * ee^T)p$$
- $\frac{1}{N} * e$  is the uniform distribution (taking the role of random teleportation)
  - what if we replaced it by something else? [that could take preferences into account]

# Personalized PageRank

$$\mathbf{p} = (\mathbf{a} * \mathbf{M} + (1-\mathbf{a}) * \frac{1}{N} \mathbf{e} \mathbf{e}^T) \mathbf{p}$$

$$= \mathbf{a} * \mathbf{M}\mathbf{p} + (1-\mathbf{a}) * \frac{1}{N} \mathbf{e} \mathbf{e}^T \mathbf{p} \quad (\mathbf{e}^T \mathbf{p} = 1 \text{ because } \mathbf{p} \text{ is a probability distribution})$$

$$= \mathbf{a} * \mathbf{M}\mathbf{p} + (1-\mathbf{a}) * \frac{1}{N} \mathbf{e} \quad (\text{replace } \frac{1}{N} \mathbf{e} \text{ by a preference distribution } \mathbf{q})$$

$$= \mathbf{a} * \mathbf{M}\mathbf{p} + (1-\mathbf{a}) * \mathbf{q}$$

- $\mathbf{q}$  is the *personalization vector* (distribution)
  - E.g., uniform over pages about data science
  - Idiosyncratic to a given user

We can still do power iteration here!

**Not more computationally expensive**

**But more flexible to "bias" PageRank**

# Distributed PageRank with Spark

- Core computation is **matrix multiplication**
  - We know how to do that with map-reduce
  - Complexity depends on network sparsity
- Also possible in Spark using the GraphX package
- High-level interface: [GraphFrames](#)
  - “GraphX is to RDDs as GraphFrames are to DataFrames.”

```
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends() # Get example graph

# Run PageRank until convergence to tolerance "tol".
results = g.pageRank(resetProbability=0.15, tol=0.01)
# Display resulting pageranks and final edge weights
# Note that the displayed pagerank may be truncated, e.g., missing the E notation.
# In Spark 1.5+, you can use show(truncate=False) to avoid truncation.
results.vertices.select("id", "pagerank").show()
```

$$\mathbf{p} \leftarrow a * \mathbf{M}\mathbf{p} + (1-a) * \mathbf{q}$$

# Next week

- Socio-cultural impact of Big Data
  - Filter bubbles
  - Polarization
  - (Differential) privacy
- After that:  
Present and future of Big Data

**Q & R**

# 12: Socio-cultural impact

DS-GA 1004: Big Data



# This week

- Socio-cultural impact of recommender / information retrieval systems
- Privacy and de-anonymization
- Differential privacy

# Socio-cultural impact of recommender and information retrieval systems

# Claim: Recommender systems might have killed Buzzfeed

TECHNOLOGY

## The Internet of the 2010s Ended Today

*BuzzFeed News* was more than a website: It defined an era.

By Charlie Warzel

AP

The Associated Press @AP · Apr 20

Pulitzer Prize-winning digital media company BuzzFeed will shut down its news division as it cuts about 15% of its entire staff. The company will now have one remaining news brand, HuffPost.



APRIL 20, 2023

SHARE

SAVED STORIES

SAVE

THE MEDIA

## BuzzFeed News Is Dead

Who is really to blame?

BY HILLARY FREY APRIL 20, 2023 • 5:16 PM



Founder and CEO of BuzzFeed Jonah H. Peretti speaks during BuzzFeed Inc.'s listing day at Nasdaq on Dec. 6, 2011. Bennett Raglin/Getty Images

# Echo chambers and filter bubbles

- Recommender systems rely on **similarity**
- Over time, this **reduces the diversity** items that a user sees

- Users tend to herd together around similar items



- This is called a **filter bubble** [Pariser, 2011]

- Best case: users get only what they like
- Typical case: user gets **bored and leaves**
- Worst case: user becomes **isolated and polarized (extreme echo chambers)**

# Targeted advertising

- Not all personalization comes through explicit interactions
- Personalized advertising platforms rely on user features
- This can get you into trouble...
  - Age?
  - Gender?
  - Zip code?



[Washington Post, 2019-03-19](#)

# Targeted advertising

Federal housing law [prohibits discrimination](#) based on race, color, religion, national origin, gender, disability or family status. Facebook said the new platform will also prevent advertisers from discriminating based on sexual orientation, age, ethnicity and other characteristics covered by state and local civil rights laws.

Facebook classified people according to their demographics, behaviors and interests using terms such as “English as a second language,” “disabled parking permit” or “Telemundo” — which critics argue are proxies for protected categories of people.

- Gender?
- Zip code?

tions  
res



[Washington Post, 2019-03-19](#)

# The ethics can be complicated: Representation

- Recommender systems depend on past behavior of the *user population*
- Dealing directly with user data can be sensitive...
  - How is your *user population biased*?
  - How does your *model handle bias*?
  - How are *atypical users* treated?
- Always remember to ask:
  - *Who benefits* from a recommender system?
  - *Is personalization needed?*

# Privacy and de-anonymization

# Open data and anonymization

- Scientific progress is built on open data, but human data is often sensitive
- Datasets are often **anonymized** by obscuring identifiers, but this may not be enough to protect privacy!
- Even statistical summaries can violate privacy

# Commonly used, but insufficient strategies for protecting users in open data

- Obfuscate identifiers:

*Jean Valjean* ⇒ 24601

10 March 2023

Some interesting discoveries in a shared dataset: Néma et al. (2022).

In this post I'm going to be discussing this article, but mostly its dataset:

Néma, J., Zdara, J., Lašák, P., Bavlovič, J., Bureš, M., Pejchal, J., & Schvach, H. (2023). Impact of cold exposure on life satisfaction and physical composition of soldiers. *BMJ Military Health*. Advance online publication. <https://doi.org/10.1136/military-2022-002237>

The first and most important problem in the file as it was uploaded, and was in place until a couple of days ago, is that a lot of PII was left in there. Specifically, the file contained *the first and last names and date of birth of every participant*. This study was carried out in the Czech

- Perturb observations:  $X[100, 'height'] += N(0, 1)$

- Can impact accuracy
  - Can sometimes be undone

- Limit per-user information

- E.g., ***k-anonymity*** [Sweeney, 2002]: each attribute occurs in at least  $k$  rows, can't distinguish  $k-1$  rows

- Statistical summaries instead of individual records

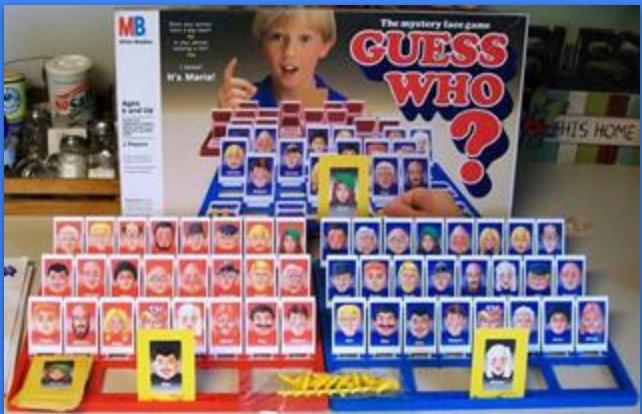
- Can still leak individual information!

# What is a de-anonymization attack?

- Say we release an “anonymized” dataset  $R = (r_1, r_2, \dots, r_N)$
- Given some partial, potentially inaccurate observations for an individual:
  - Can we determine if the individual was included in  $R$ ?
  - Can we recover more information about the individual (that was not released)?
- Narayanan and Shmatikov (2008) demonstrated that this is possible in the Netflix prize dataset!



# Why is k-anonymity not enough?



- If each attribute occurs in at least  $k$  rows, then **no single attribute is informative**
- But **combinations of attributes are often informative!**
- In large, high-dimensional collections, **this is the rule, not the exception!**
- People are ultra-high-dimensional, and highly idiosyncratic, which is often reflected in data about people.

# List of personal attributes (there are others)

1. Height
2. Date of birth
3. Gender
4. Race
5. Sexual orientation
6. Eye color
7. Hair color
8. Chronic medical conditions
9. Education
10. Sibship status
11. Location of birth (zip code, if possible)
12. Location of current residence (zip code, if possible)
13. Marital status
14. Number of children
15. Job title
16. Approximate household income
17. Homeowner or renter
18. Primary language spoken at home
19. Religious affiliation, if any
20. Citizenship

# Deanonymizing the Netflix prize: How the attack worked

[Narayanan & Shmatikov, 2008]

- Step 1: define similarity between rows as **average similarity** over their **support**

$$\text{Sim}(R_u, R_v) = [\sum_i \text{Sim}(R_{ui}, R_{vi})] / |R_u \cup R_v|$$

- Step 2: Given a partial observation  $q$ , compute similarity to each row.
- Step 3: Determine a threshold by comparing top score to second highest score
  - If sufficiently large difference, report a match, otherwise, report no match
- **Question: how much partial data do we need to succeed?**

# The similarity measures

- Per-user similarity:  $\text{Sim}(R_u, R_v) = [\sum_i \text{Sim}(R_{ui}, R_{vi})] / |R_u \cup R_v|$
- Boolean **observation similarity** if ratings are within a threshold:
  - $|R_{ui} - R_{vi}| = 0$  (identical ratings)
  - $|R_{ui} - R_{vi}| \leq 1$  (off by at most 1 star)
- For timestamps, thresholds of 3 or 14 days
- Matching threshold:  $\text{Sim}(q, R_1) - \text{Sim}(q, R_2) > 1.5 \cdot \sigma_w(\text{Sim}(q, R_w))$

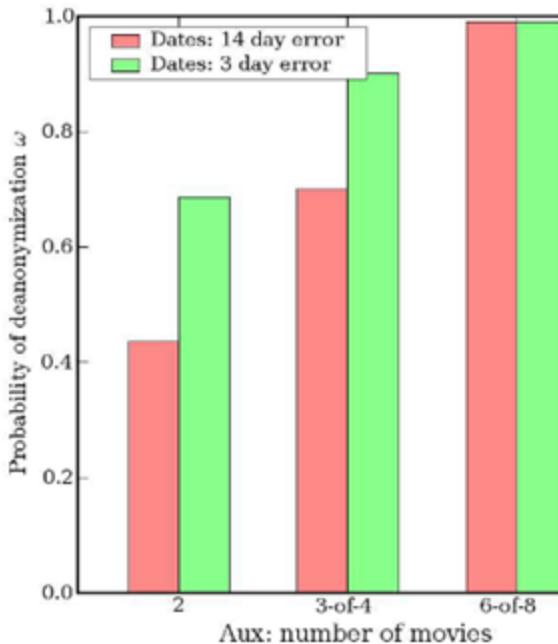


Figure 1: De-anonymization: adversary knows exact ratings and approximate dates.

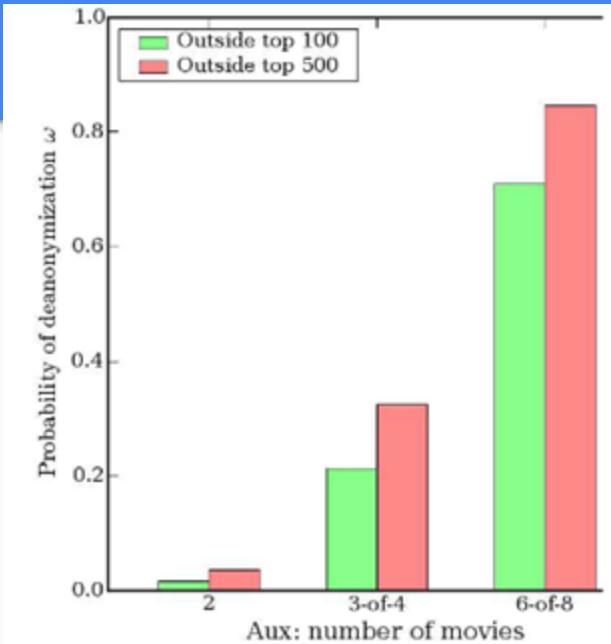


Figure 4: Adversary knows exact ratings but does not know dates at all.

With 8 ratings (2 may be wrong) and 14-day error on rating date, 99% of records can be uniquely identified!

Even without timestamps, unpopular items are highly informative!

**How big of a deal is the fact that almost all records could be uniquely identified in this way?**

Be honest

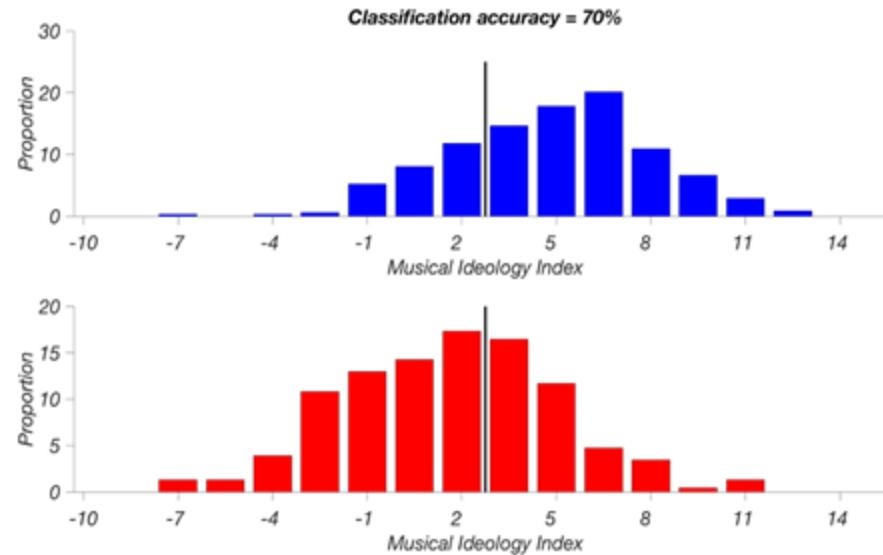
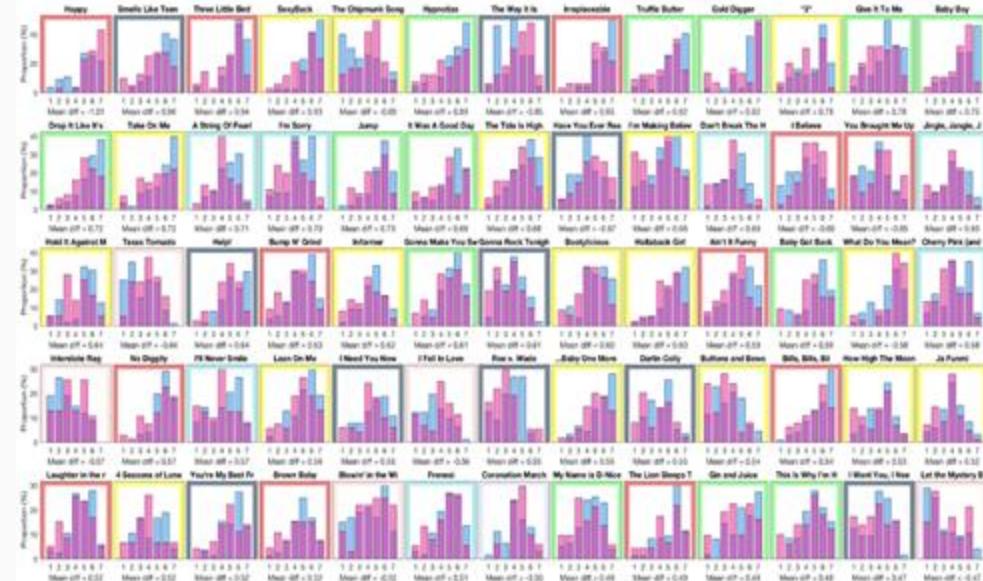
# Why does this matter?

- Some comments from Narayanan and Shmatikov:
  - “The privacy question is *not* ‘Does the average Netflix subscriber care about the privacy of his movie viewing history?’ but ‘Are there *any* Netflix subscribers whose privacy can be compromised by analyzing the Netflix Prize dataset?’”
  - Entertainment preferences can correlate with personal information (political leanings, sexual orientation, religious affiliation, etc...)
  - “The simple-minded division of information into **personal** and **non-personal** is a false dichotomy.”
- Breaches are **irrevocable** and may have implications for a **user's future privacy**

# It might be - there are all kinds of interesting correlations...

## *The Songs of My People: Appraisal Differences of Popular Music as a Function of Ideology*

Myers, C.F., Spivack, S., Spilka, N.H., Philibotte, S.J., Passman, I.J., & Wallisch, P.



## What we did

- Database reconstruction for all 308,745,538 people in 2010 Census
- Link reconstructed records to commercial databases: acquire PII
- Successful linkage to commercial data: putative re-identification
- Compare putative re-identifications to confidential data
- Successful linkage to confidential data: confirmed re-identification
- Harm: attacker can learn self-response race and ethnicity

## What we found

- Census block and voting age (18+) correctly reconstructed in all 6,207,027 inhabited blocks
- Block, sex, age (in years), race (OMB 63 categories), ethnicity reconstructed
  - Exactly: 46% of population (142 million of 308,745,538)
  - Allowing age +/- one year: 71% of population (219 million of 308,745,538)
- Block, sex, age linked to commercial data to acquire PII
  - Putative re-identifications: 45% of population (138 million of 308,745,538)
- Name, block, sex, age, race, ethnicity compared to confidential data
  - Confirmed re-identifications: 38% of putative (52 million; 17% of population)
- For the confirmed re-identifications, race and ethnicity are learned correctly, although the attacker may still have uncertainty

# Tracking user behavior / recommender systems as potential privacy violations

About a year after Pole created his pregnancy-prediction model, a man walked into a Target outside Minneapolis and demanded to see the manager. He was clutching coupons that had been sent to his daughter, and he was angry, according to an employee who participated in the conversation.

“My daughter got this in the mail!” he said. “She’s still in high school, and you’re sending her coupons for baby clothes and cribs? Are you trying to encourage her to get pregnant?”

# Tracking user behavior

“With the pregnancy products, though, we learned that some women react badly,” the executive said. “Then we started mixing in all these ads for things we knew pregnant women would never buy, so the baby ads looked random. We’d put an ad for a lawn mower next to diapers. We’d put a coupon for wineglasses next to infant clothes. That way, it looked like all the products were chosen by chance.

# Tracking user behavior: Pulling back

- Recommenders rely on past interactions to predict future behavior
- Target's system *inferred and disclosed* pregnancy to a teenager's family
- *Later*: personalized ads were obfuscated and made to look *random*

**Do you think Target's use of collaborative filtering was a violation of privacy?**

**Why?**

# Now



- Socio-cultural impact of recommender systems
- Privacy and de-anonymization
- **Differential privacy**

*Cynthia Dwork*

# What are we exposing to the world?

- A whole dataset (raw data)?
- A “de-identified” dataset?
- A fixed set of statistics measured from a dataset?
- A statistical model derived from a dataset?
- An API to ask questions of a dataset?

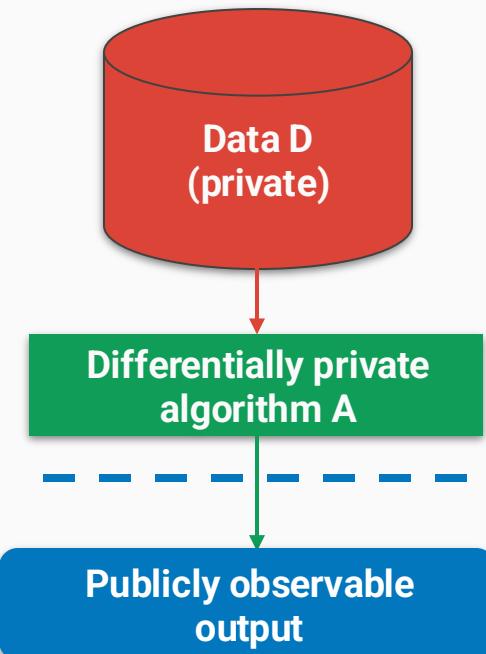
# Differential privacy

[Dwork, McSherry, Nissim, Smith, 2006]

- High-level idea:

If an individual is **excluded from the data**,  
the result of a computation **should not change**

- We'll achieve this by **randomizing** the computation
- DP is a property of an **algorithm**, not the data!



# Differential privacy (definition)

[Dwork, McSherry, Nissim, Smith, 2006]

For any two datasets  $D$  and  $D'$  differing by one row (i.e.,  $D' = D + \{x\}$ ),  
a **(stochastic/randomized) algorithm  $A$**  is  $\varepsilon$ -differentially private if

$$\Pr[A(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[A(D') \in S]$$

for any  $S \subseteq \text{range}(A)$  (subset of possible outcomes of  $A$ )

# Differential privacy (definition)

[Dwork, McSherry, Nissim, Smith, 2006]

For any two datasets  $D$  and  $D'$  differing by one row (i.e.,  $D' = D + \{x\}$ ),  
a **(stochastic/randomized) algorithm  $A$**  is  $\varepsilon$ -differentially private if

$$\Pr[A(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[A(D') \in S]$$

for any  $S \subseteq \text{range}(A)$  (subset of possible outcomes of  $A$ )

Definition uses subsets  $\Pr[A(D) \in S]$  instead of specific values  $\Pr[A(D) = v]$  so that we can support densities and continuous R.V.s.

If that's weird to you, just think of  $S$  as a small interval around a specific value  $v$ .

# Differential privacy (definition)

[Dwork, McSherry, Nissim, Smith, 2006]

For any two datasets  $D$  and  $D'$  differing by one row (i.e.,  $D' = D + \{x\}$ ),  
a **(stochastic/randomized) algorithm  $A$**  is  $\varepsilon$ -differentially private if

$$\Pr[A(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[A(D') \in S]$$

for any  $S \subseteq \text{range}(A)$  (subset of possible outcomes of  $A$ )

**Intuition:**

If we observe an output value  $v$ , we can't reliably tell if it  
was computed by  $v = A(D)$  or  $v = A(D')$ .

# What does $\epsilon$ (epsilon) mean here?

- Differential privacy says:

$$\Pr[A(D) \in S] \leq \exp(\epsilon) \cdot \Pr[A(D') \in S]$$

- If  $\epsilon$  is small (near 0), then  $\exp(\epsilon)$  is near  $\exp(0) = 1$ , and we'll have

$$\Pr[A(D) \in S] \lesssim \Pr[A(D') \in S]$$

- DP is a *symmetric* definition, do we'd also have the reverse:

$$\Pr[A(D) \in S] \gtrsim \Pr[A(D') \in S]$$

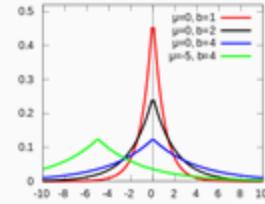
- When  $\epsilon$  is large, the bounds are looser

# Tuning the noise:

How much noise do we need to add to ensure privacy?  
Sensitivity and the Laplace mechanism

- Say we have a vector-valued, private function  $f: D \rightarrow \mathbf{R}^d$ 
  - Example: average across rows of some real-valued attributes
- How different are  $f(D)$  and  $f(D')$  if  $D$  and  $D'$  differ by a single row?
  - Define this as the **sensitivity** of  $f$
- Let  $A(D) = f(D) + z$  where  $z[i] \sim \text{Laplace}(0, \Delta f / \varepsilon)$ .  
Then  $A$  is  $\varepsilon$ -differentially private!

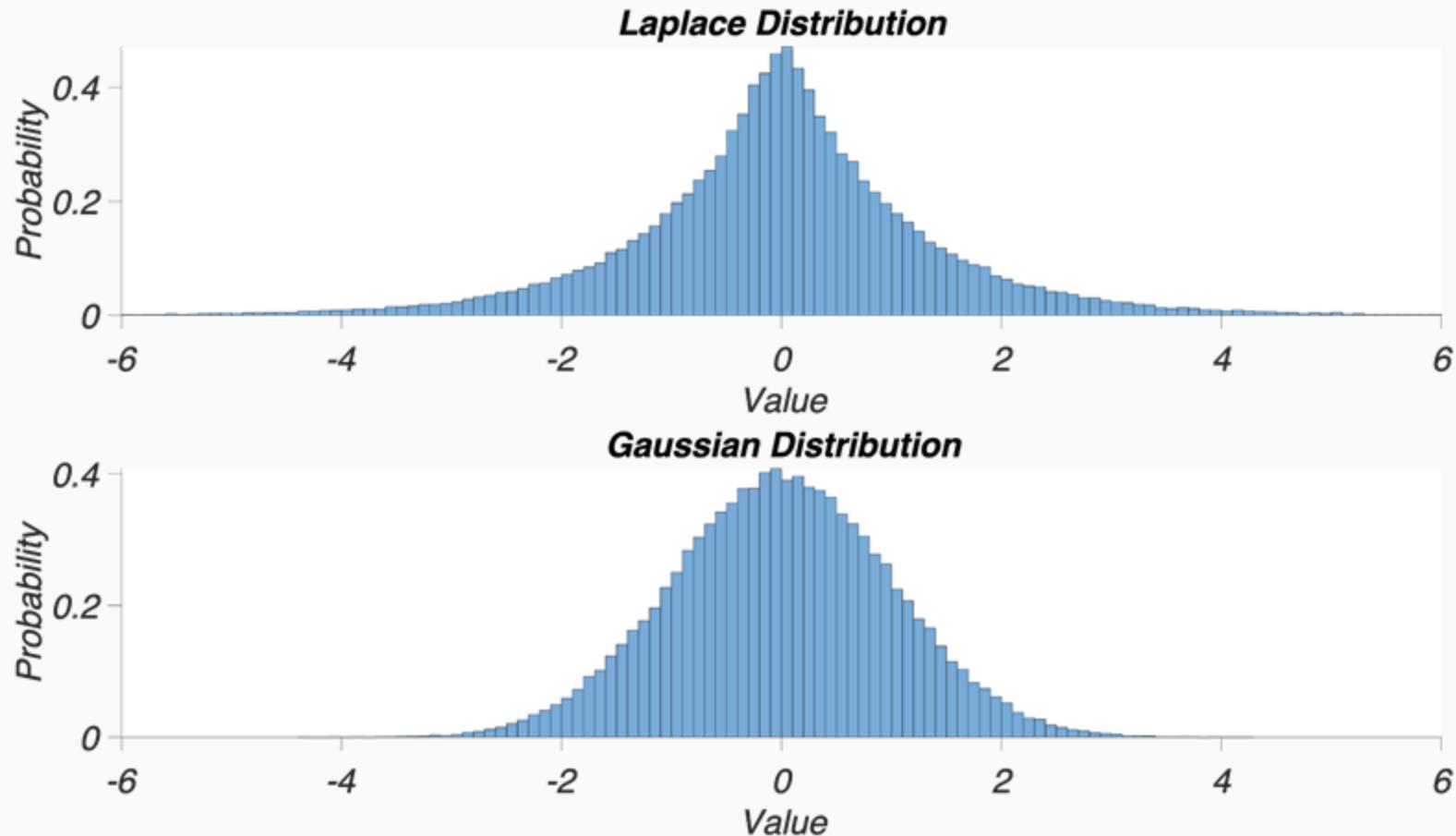
$$\Delta f = \max_{D, D'} \sum_i |f(D)[i] - f(D')[i]|$$



$z \sim \text{Laplace}(\mu, b)$ :

$$p(z) = \exp(-|z-\mu|/b) / 2b$$

## Brief aside: What's the right kind of noise to add?



# Aside: Be careful with “AI” (a paradigmatic case of hallucinations I saw in the wild)

What can you tell me about the laplace distribution, relative to the gaussian distribution?

wait, i thought the gaussian is falling off exponentially. is this not true?

The Laplace distribution and the Gaussian (or normal) distribution are both continuous probability distributions that describe the behavior differences.

Laplace distribution:

The Laplace distribution, also known peakedness at the mean and faster

$$f(x | \mu, b) = (1 / 2b) * \exp(-|x - \mu| / b)$$

Here,  $\mu$  is the location parameter (mean), symmetric about the mean, and its tails decay more slowly compared to the Gaussian distribution:

The Gaussian distribution, also known curve, with the mean, median, and mode being the same.

$$f(x | \mu, \sigma) = (1 / (\sigma * \sqrt{2 * \pi})) * \exp(-\frac{(x - \mu)^2}{2\sigma^2})$$

Here,  $\mu$  is the mean and  $\sigma > 0$  is the standard deviation. The Gaussian distribution is also symmetric about the mean, but its tails decay more slowly compared to the Laplace distribution.

Relative comparison:

1. Tails: The Laplace distribution has heavier tails, which means it assigns more probability to extreme values compared to the Gaussian distribution. The tails of the Laplace distribution decay exponentially, whereas the tails of the Gaussian distribution decay quadratically.

The laplace distribution has heavier tails than the gaussian distribution, however

Apologies for the confusion. I understand why my previous response might have been misleading. The Gaussian distribution does indeed have tails that fall off exponentially, but more specifically, they fall off as a squared exponential. To clarify, let's revisit the probability density function (PDF) of the Gaussian

$$\sqrt{2 * \pi}) * \exp(-(x - \mu)^2 / (2 * \sigma^2))$$

From this formula, the exponent in the Gaussian distribution is  $-(x - \mu)^2 / (2 * \sigma^2)$ , which is a squared term of  $(x - \mu)$ . This means that the tails of the Gaussian distribution fall off as a squared exponential, which is slower than the simple exponential decay of the Laplace distribution's tails. In contrast, the Gaussian distribution's tails do fall off exponentially but as a squared exponential, which is even slower than the exponential decay of the Laplace distribution's tails.

What do you mean slower? It's much faster. The laplace distribution has heavier tails

I'm sorry, and I apologize for the confusion in my previous response. The tails of the Laplace distribution indeed decay faster than those of the Gaussian distribution.

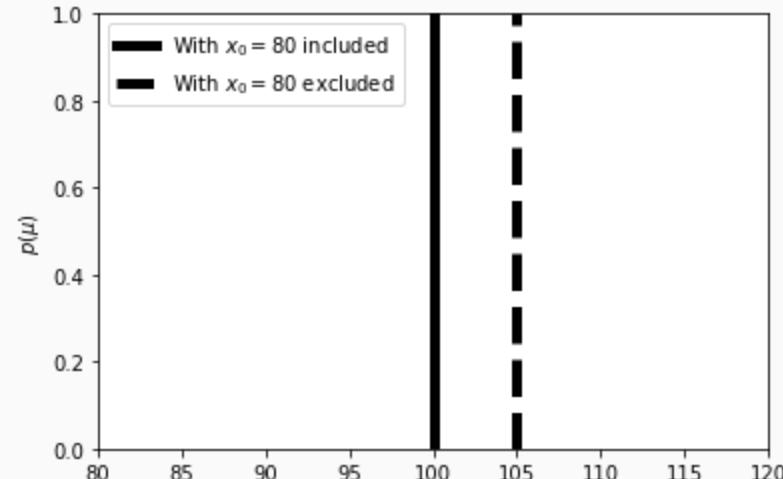
More precisely:

- The Laplace distribution's tails decay at a rate proportional to  $\exp(-|x - \mu| / b)$ . This is a simple exponential decay, which makes the tails of the Laplace distribution lighter and decay faster than the Gaussian distribution's tails.
- The Gaussian distribution's tails decay at a rate proportional to  $\exp(-(x - \mu)^2 / (2 * \sigma^2))$ . This is a squared exponential decay, which results in slower decay and heavier tails compared to the Laplace distribution.

Thank you for pointing out the inconsistency, and I apologize for any confusion that may have arisen. To summarize, the Laplace distribution has faster decaying, lighter tails than the Gaussian distribution.

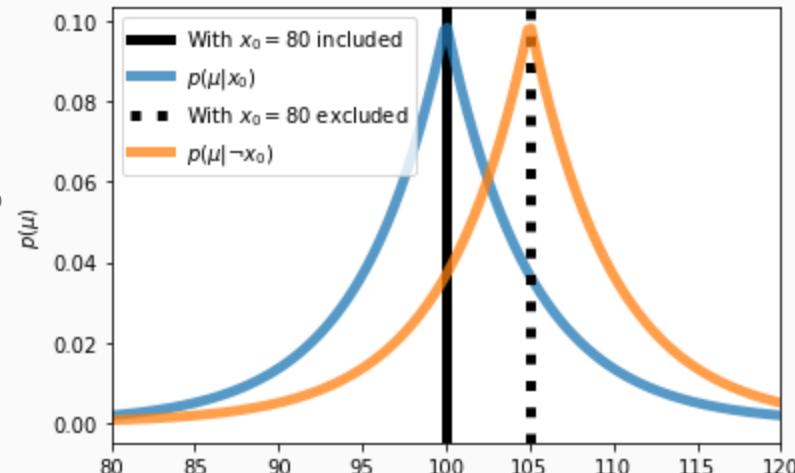
# Why is adding Laplacian noise reasonable?

- Say we have data  $X = [80, 90, 100, 110, 120]$
- We want to compute the mean
- What happens if  $x_0=80$  is included or not?



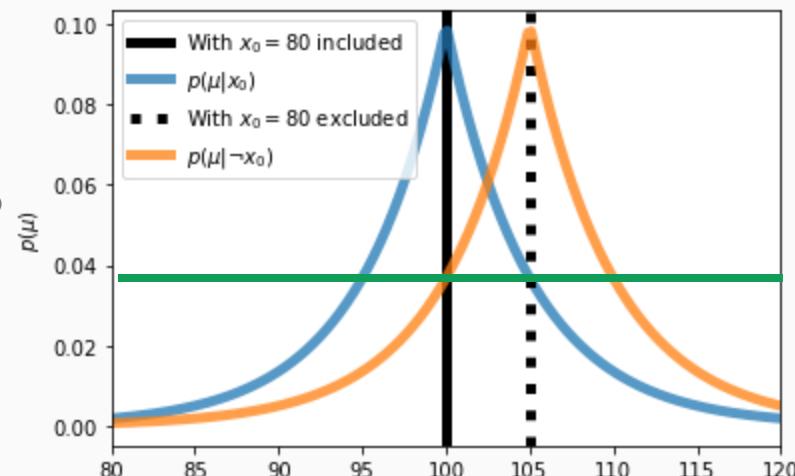
# Why is adding Laplacian noise reasonable?

- Say we have data  $X = [80, 90, 100, 110, 120]$
- We want to compute the mean
- What happens if  $x_0=80$  is included or not?



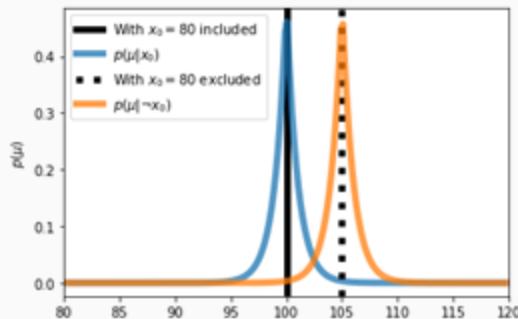
# Why is adding Laplacian noise reasonable?

- Say we have data  $X = [80, 90, 100, 110, 120]$
- We want to compute the mean
- What happens if  $x_0=80$  is included or not?
- If we observe  $A(D) = 105$ :
  - Not much evidence that  $x_0$  was (not) included!

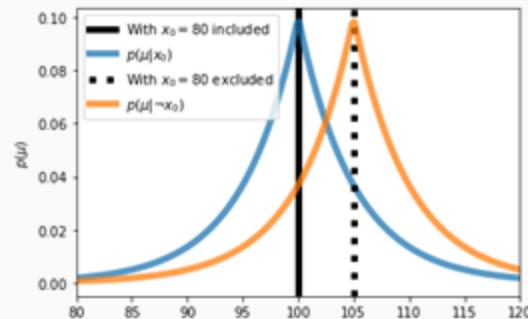


High noise  $\Rightarrow$  high privacy, but there are tradeoffs, as it lowers accuracy

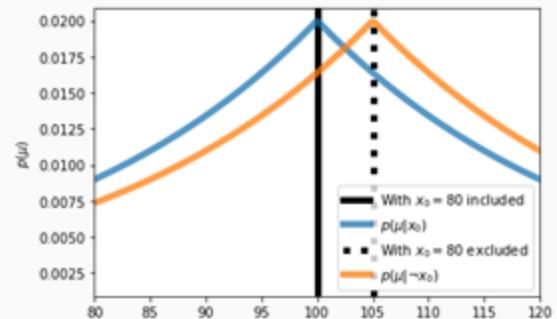
Low noise ( $b=1$ )



Medium noise ( $b=5$ )



High noise ( $b=25$ )



Large  $\epsilon \Rightarrow$  High accuracy

$z \sim \text{Laplace}(\mu, b)$ :

$$p(z) = \exp(-|z-\mu|/b) / 2b$$

Small  $\epsilon \Rightarrow$  Low accuracy

# Example of differential privacy in action: counts

- Say our data are binary vectors in  $\mathbb{R}^5$ , e.g.  $x_i = (0, 1, 1, 0, 0)$
- Let  $f(D)$  be the sum over rows in  $D$ . What's the sensitivity?
  - Let  $D' = D + \text{one row } x_N$
  - For each attribute  $i$ , we have
$$f(D')[i] = f(D)[i] + x_N[i] \Rightarrow |f(D')[i] - f(D)[i]| = |x_N[i]| \leq 1$$
  - So for 5 attributes, we have a **sensitivity of 5**
- To get  $\epsilon$ -privacy, we need Laplace noise of scale  $5/\epsilon$

Say  $D$  has 50 records, each  $x$  in  $D$  is a single number between -10 and 10, and  $f(D) = \max(\text{abs}(x))$   
Sensitivity:  $f(D) = \max(|x|)$

- If  $x \sim [-10, 10]$ , then  $|x| \in [0, 10]$
- What's the biggest gap that  $f(D)$  and  $f(D')$  could have?
  - Worst case:  $D = [0, 0, 0, 0, \dots]$ , and  $D' = D + \{10\}$
  - $f(D) = 0, f(D') = 10$
- Sensitivity  $\Delta f = |10 - 0| = 10$
- Take-away, extremal aggregators (min, max) are very sensitive!
  - Safer alternatives exist, eg 1%- and 99%-percentile

In contrast: Same setting as before, but now the aggregator is the mean

Sensitivity:  $f(D) = \text{mean}(x)$ ,  $N = 50$

- If  $x \sim [-10, 10]$ , then  $\text{mean}(x) \in [-10, 10]$
- $f(D) = \sum x / (N-1)$   
$$f(D') = [(N-1)f(D) + x_N] / N$$
- $|f(D') - f(D)| = |[(N-1)f(D) + x_N] / N - f(D)|$   
$$= |[(N-1)f(D) + x_N] - Nf(D)| / N$$
  
$$= |x_N - f(D)| / N$$

# Sensitivity, sample size, scale and privacy

- If  $x \sim [-10, 10]$ , then  $\text{mean}(x) \in [-10, 10]$

- $f(D) = \sum x / (N-1)$

$$f(D') = [(N-1)f(D) + x_N] / N$$

- $|f(D') - f(D)| = |[(N-1)f(D) + x_N] / N - f(D)|$   
 $= |[(N-1)f(D) + x_N] - Nf(D)| / N$   
 $= |x_N - f(D)| / N$   
 $\leq |10 - -10| / N$   
 $= 20 / N$

Sensitivity decreases as  $N$  increases!

Privacy is easier to achieve in larger datasets.



In small samples,  
Waldo is easy to find

WHERE'S  
WALDO?

# What about multiple queries?

- Each time you ask for  $A(\cdot)$ , you'll get a different answer (it's randomized!)
- Differential privacy **composition** theorem:

If you make a sequence of queries  $A_i$ , each being  $\varepsilon_i$ -DP, then the result is only guaranteed to be ( $\bar{\varepsilon} = \sum_i \varepsilon_i$ )-differentially private

- The good news: any **deterministic** post-processing preserves privacy!

## Extra resources

-  [Cynthia Dwork & Aaron Roth \(2014\)](#)  
[monograph on Differential Privacy](#)
-  [Kamalika Chaudhuri & Anand Sarwate \(2017\)](#)  
[NeurIPS tutorial on Differentially Private Machine Learning](#)
-  [John Abowd \(2019\)](#)  
[ICML Keynote on the 2020 US Census](#)
-  [danah boyd & Jayshree Sarathy \(2022\)](#)  
[Summary / updates to ongoing DP controversy w/ 2020 Census](#)

# Differential privacy: Summary

- Being able to release data is critical for **reproducibility** but needs to be balanced with concerns about **privacy**.
- Removing **identifiers** and providing **k-anonymity** is not sufficient to guarantee privacy, particularly if the data is high-dimensional.
- In that case, **de-anonymization attacks** are highly accurate, even if based only on crude similarity search and relatively little data (e.g. the Netflix prize attack)
- **Differential privacy** keeps the raw data private, but lets the user interact with it via an **API**.
- Idea: Add carefully calibrated **Laplacian** noise to the data returned to give plausible deniability (as to whether the results came from a dataset containing a given row)
- **Sensitivity**: Maximal difference in output given a single row difference.  
Extremal aggregators are highly sensitive, but higher N reduces sensitivity.
- **Multiple queries** reduce differential privacy - privacy loss aggregates.