Smallest font

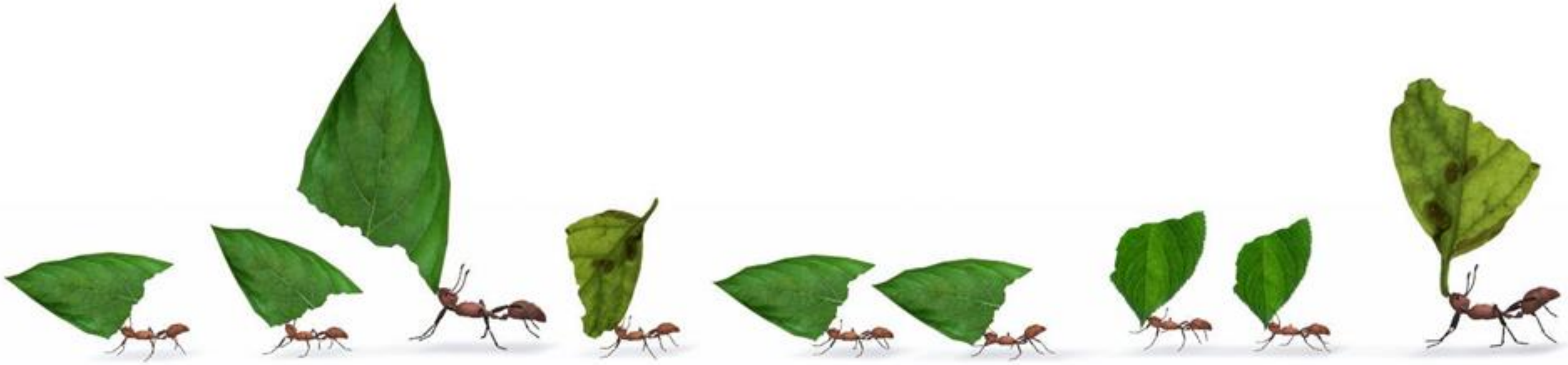
Please turn off and put away your cell phone

Calibration slide

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

Smallest font

Big Data



NYU | Center for Data Science

DS-GA 1004: Big Data

Week 04: Distributed storage

HDFS

Remember the dataphant?

IDS:
Statistics
crash course
(filtered by DS relevance)

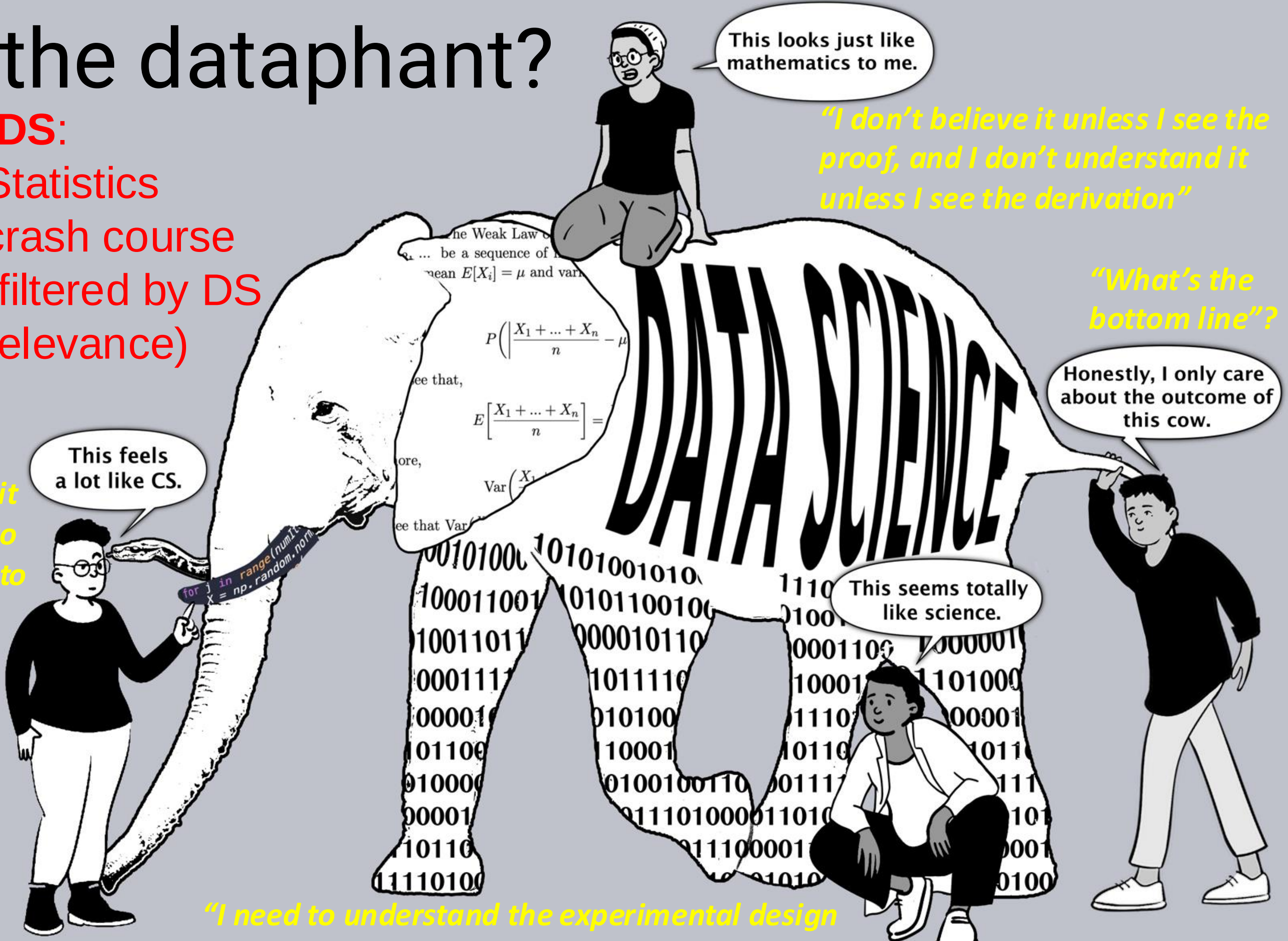
Big Data:
CS
crash course
(filtered by DS relevance)

"I don't believe it unless I see the proof, and I don't understand it unless I see the derivation"

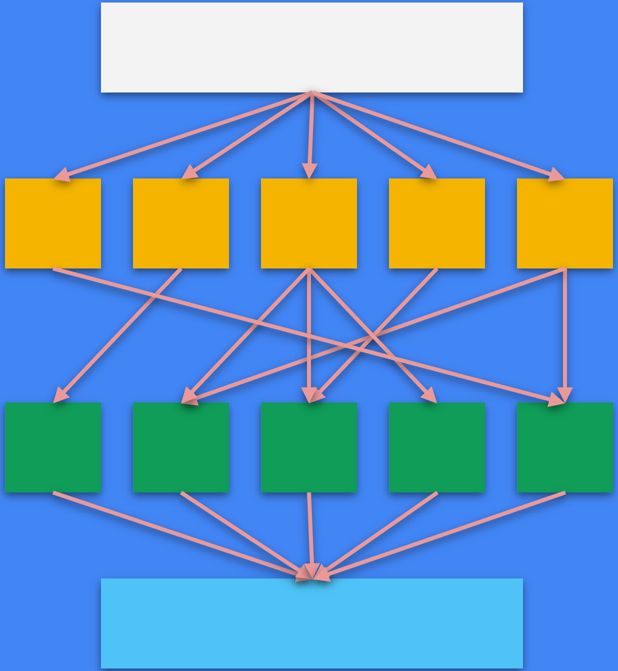
"What's the bottom line"?

"I don't believe it unless I implement it in code and I need to grok the algorithm to understand it"

"I need to understand the experimental design and see the data before I believe anything"


This looks just like mathematics to me.
DATA SCIENCE
Honestly, I only care about the outcome of this cow.
This feels a lot like CS.
This seems totally like science.

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 $f(x) = x \bmod 10$

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

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

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

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

Is f(x) a hash function?

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

$g(7) = 26$

$g(42) = 34$

$g(420) = 04$

$g(2777) = 97$

Is g(x) a hash function?

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?

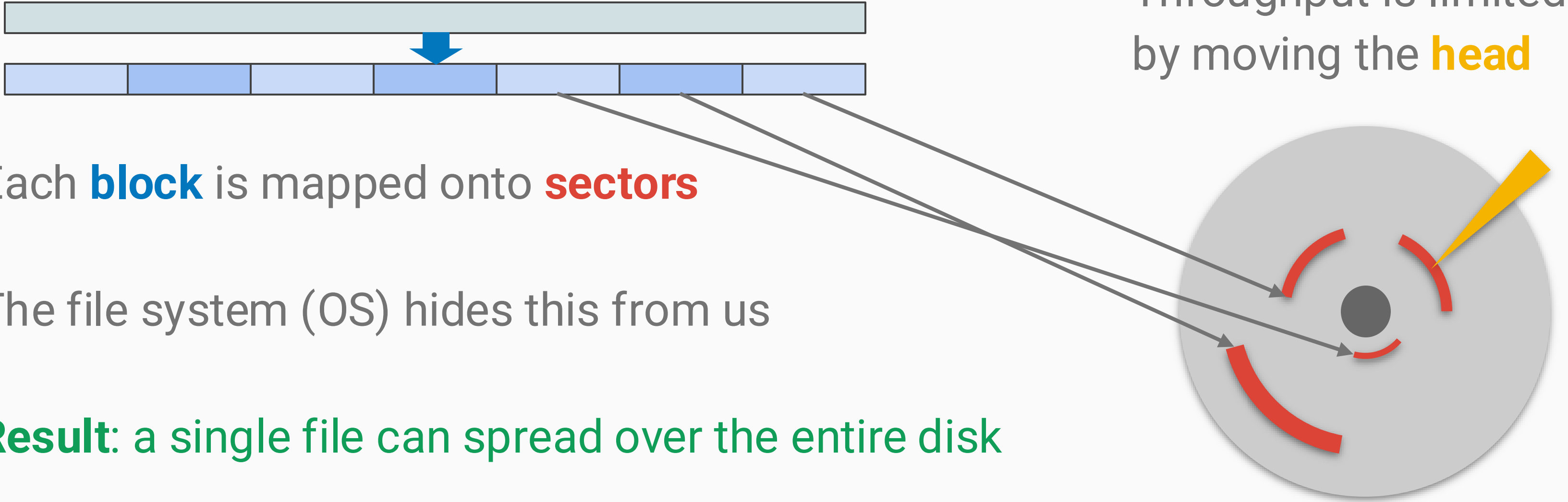
Floppy disk

Hard disk

(Spinning) platter/disk

Files and blocks

- **Files** are broken into **blocks**
  - Block size ≥ sector size



Throughput is limited by moving the **head**

- Each **block** is mapped onto **sectors**
- The file system (OS) hides this from us
- **Result**: a single file can spread over the entire disk

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

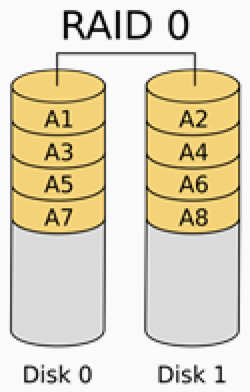
Redundant array of inexpensive disks (RAID)

- 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

[Patterson, Gibson, & Katz, SIGMOD 1988]

Commonly used RAID levels



Striping only

No fault-tolerance

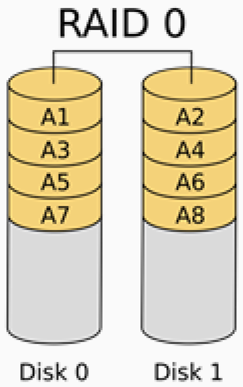
Capacity scales linearly

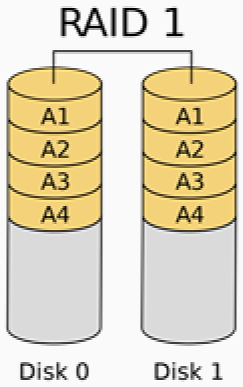
Figures from https://en.wikipedia.org/wiki/Standard_RAID_levels

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(failure/year) = 10

Commonly used RAID levels





Striping only

No fault-tolerance

Capacity scales linearly

Full redundancy

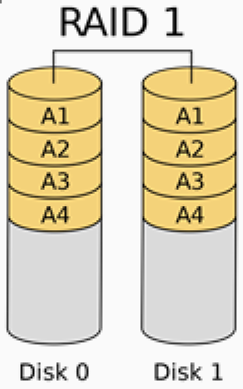
(n-1) fault-tolerance

Capacity is constant

Figures from https://en.wikipedia.org/wiki/Standard_RAID_levels

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 (⊕) :
- Say disk 0 goes down. Can we recover its data from parity?
- This logic extend beyond two disks, e.g. here for 3 disks:

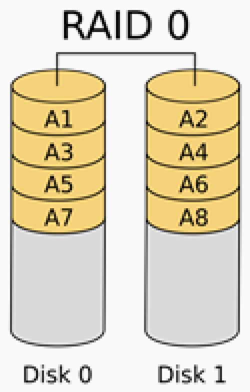
Disk 0	Disk 1	0 ⊕ 1
0	0	0
0	1	1
1	0	1
1	1	0

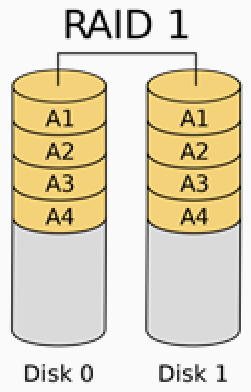
Disk 0	Disk 1	Disk 2	0 ⊕ 1 ⊕ 2
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

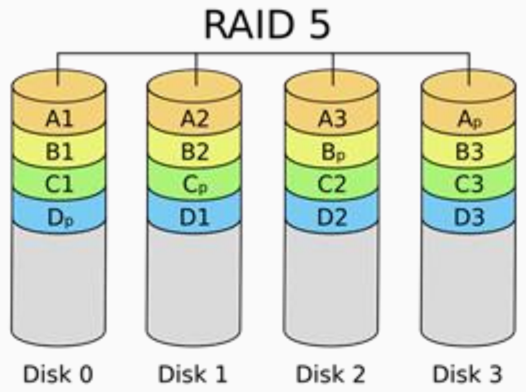
What if any one disk goes down?

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**

$A_p = A_1$ XOR A_2 XOR … XOR A_{n-1}

Can tolerate some failure

Capacity scales almost linearly

Requires n ≥ 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
Citizen	1
Resident	2
Other	3

OHE

Immigration status	Citizen?	Resident?	Other?
No	0	0	0
Yes	1	1	1

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
Citizen	1
Resident	2
Other	3

OHE

Immigration status	Citizen?	Resident?
No	0	0
Yes	1	1

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

A system of linear equations:

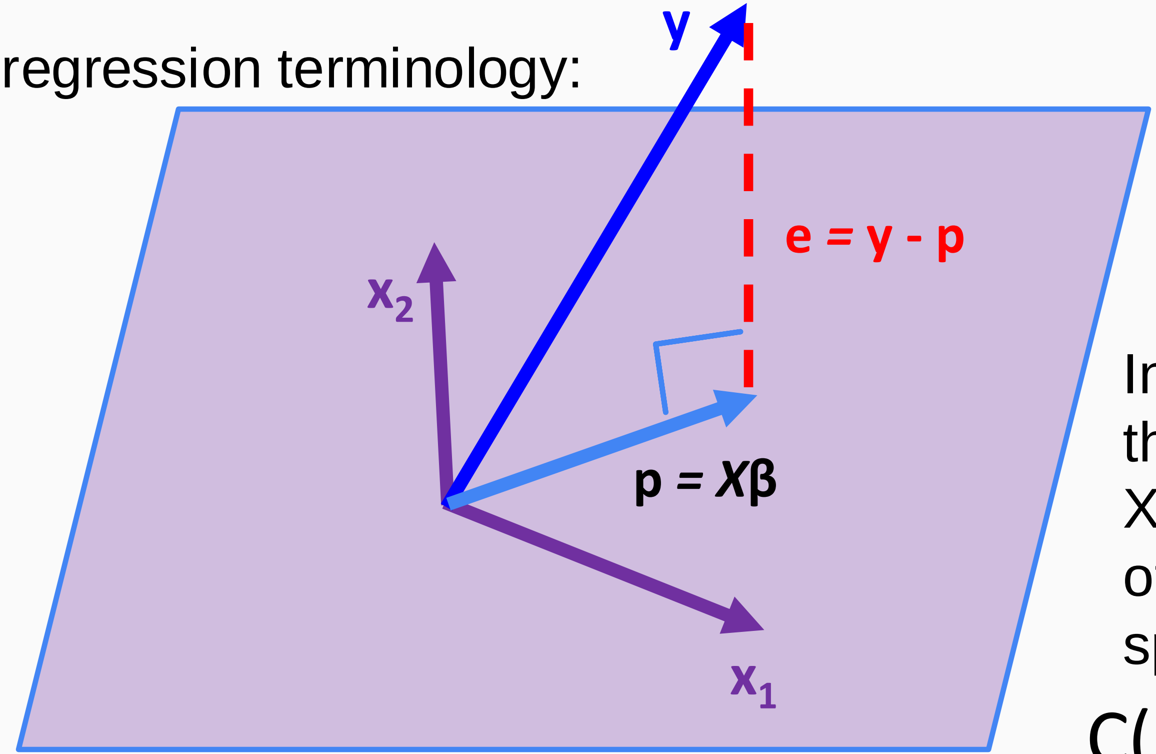
$\boldsymbol{A}\mathbf{x} = \mathbf{b}$

In linear regression terminology:

$\boldsymbol{X}\boldsymbol{\beta} = \mathbf{y}$

$\boldsymbol{X}\boldsymbol{\beta} - \mathbf{y} = \mathbf{0}$

$\boldsymbol{X} = [\boldsymbol{x}_1\ \boldsymbol{x}_2]$



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

C($\boldsymbol{X}$)

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

A system of linear equations:

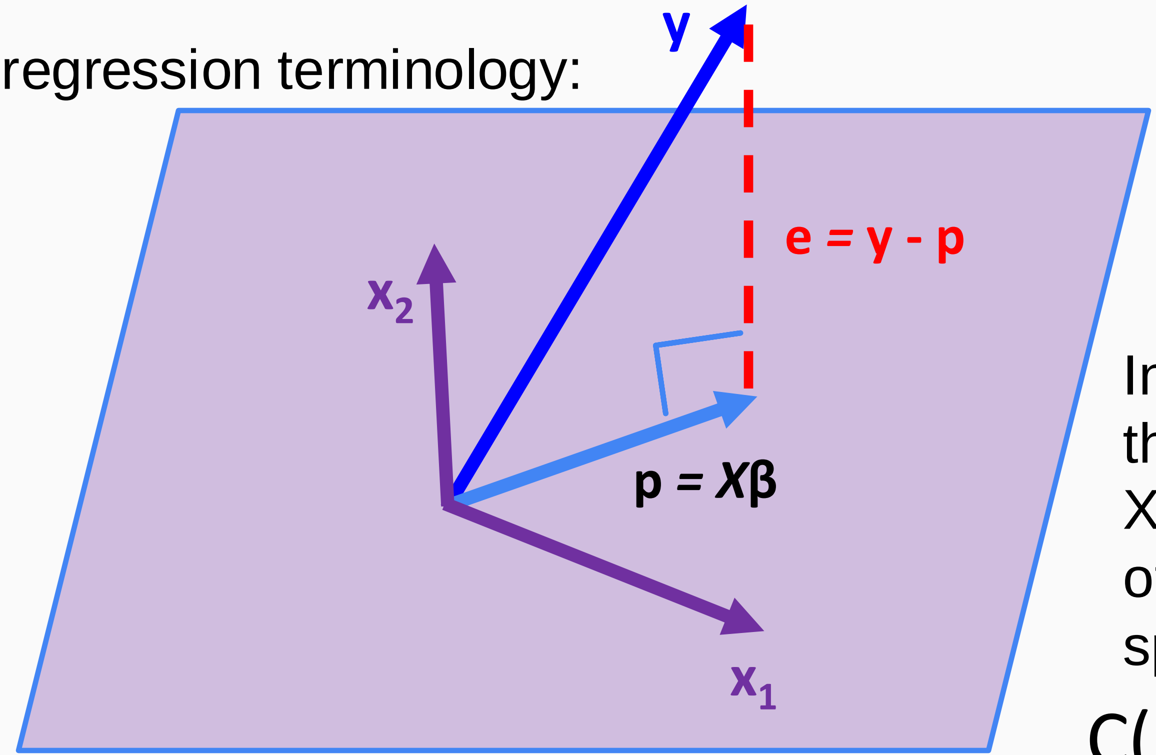
$\boldsymbol{A}\mathbf{x} = \mathbf{b}$

In linear regression terminology:

$\boldsymbol{X}\boldsymbol{\beta} = \mathbf{y}$

$\boldsymbol{X}\boldsymbol{\beta} - \mathbf{y} = \mathbf{0}$

$\boldsymbol{X} = [x_1\ x_2]$

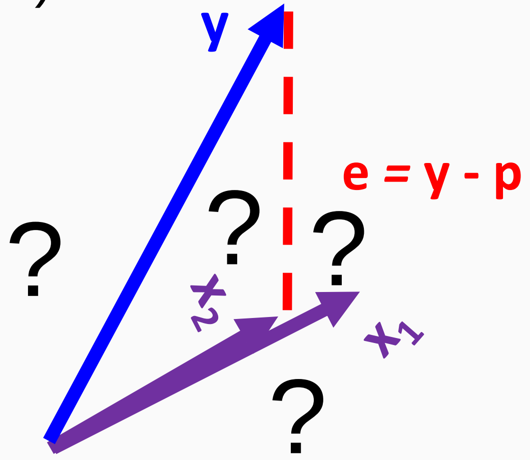


Insight: y is not in the column space of X. But the projection of y onto the column space is.

$C(\boldsymbol{X})$

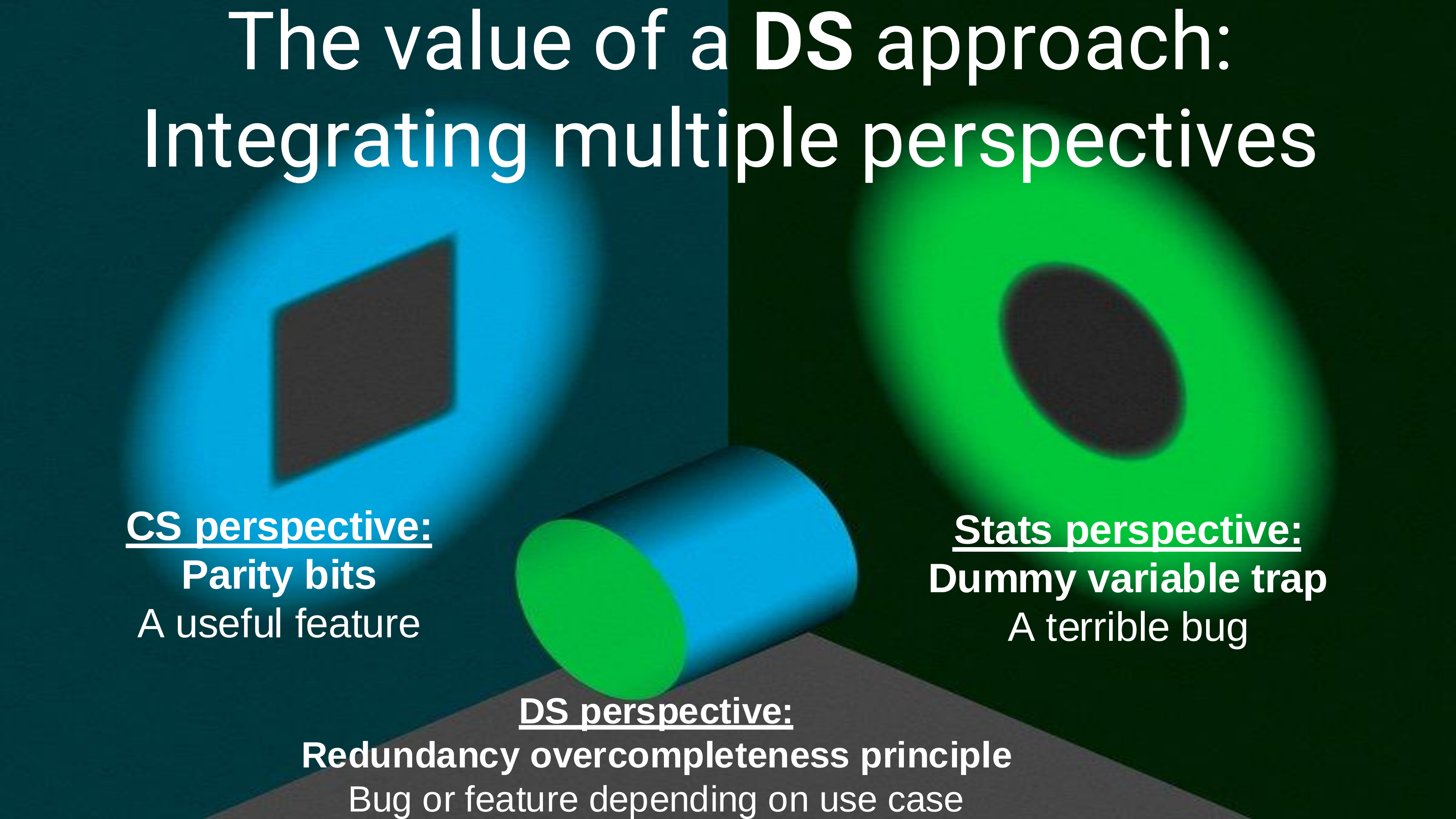
If x_1 and x_2 are pointing in similar directions, projecting on a plane becomes "wobbly" - the β estimates become unreliable:

Estimating the βs 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

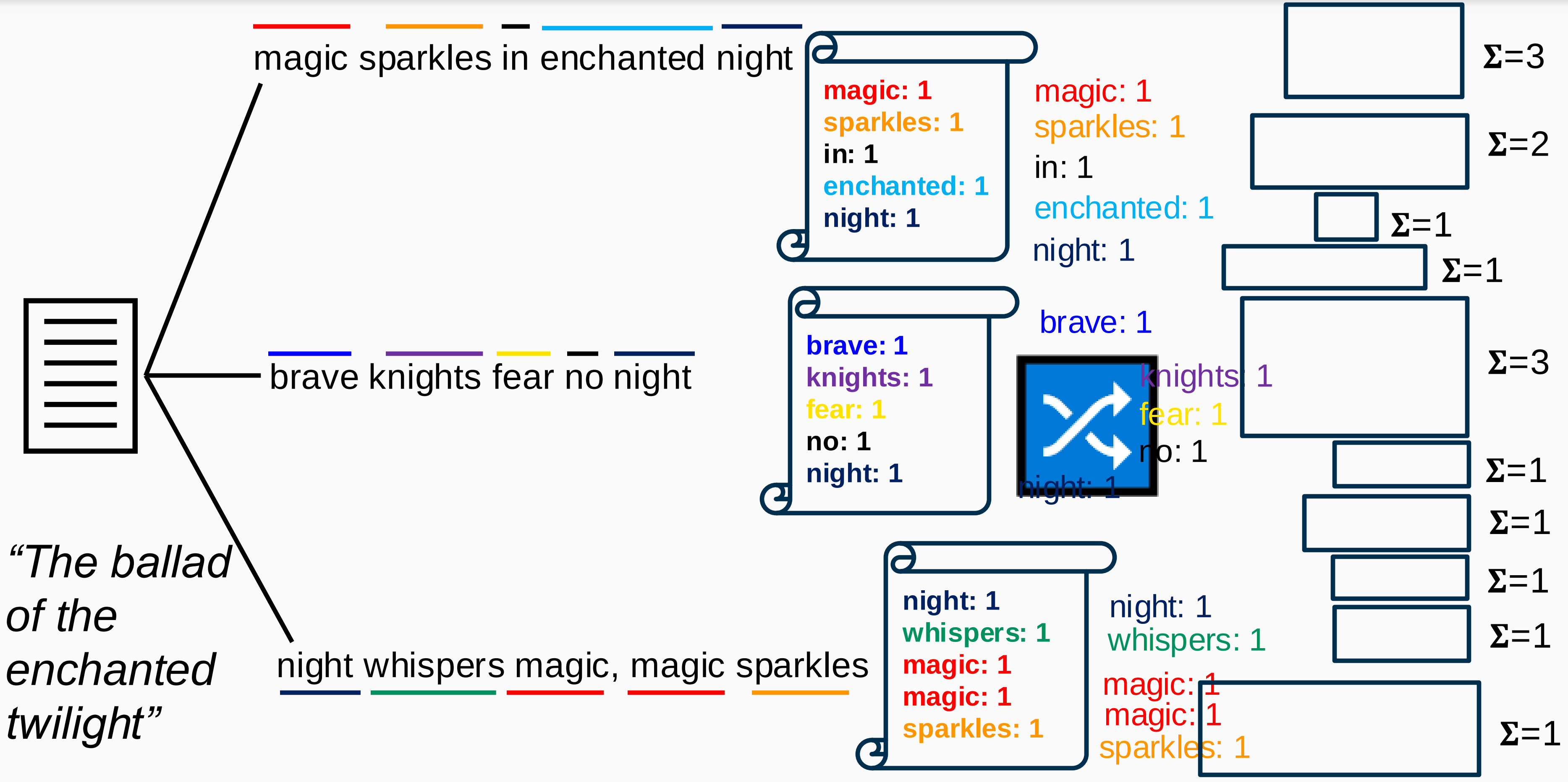
Why wasn't RAID enough?

- RAID improves capacity, fault-tolerance, and (read) throughput **on a single machine**

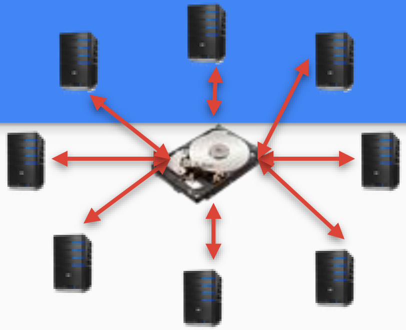
- What about distributed *computation*?
  - **Communication** over the network is a **bottleneck**

- *What are the common access patterns?*
  - Can we do **better** than both **fully localized** and **fully distributed**?

Reminder: How mapReduce works for word counts

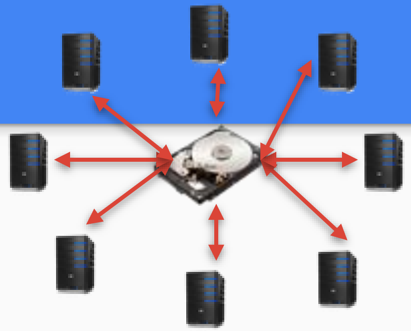


Let’s start simple



- 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



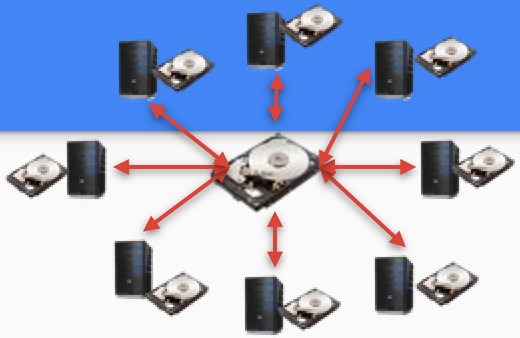
- Imagine implementing Map-Reduce from scratch
  - … with all data located on a central file server (“center node”)
- Center node sends (**mapper**, **reducer**) code to each processor **+ block of data**
- Processors send **output** back to center
  - **Mappers** → intermediate results
  - **Reducers**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

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

Each job moves the entire data set over the network!

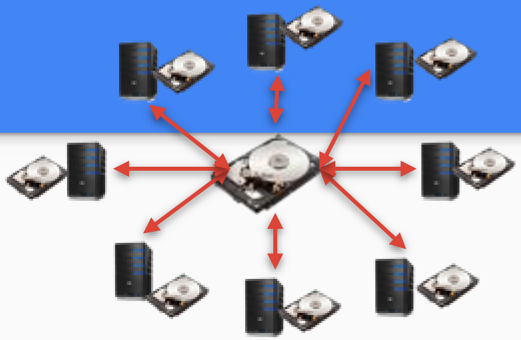
This is a **failure of data locality**.

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**)

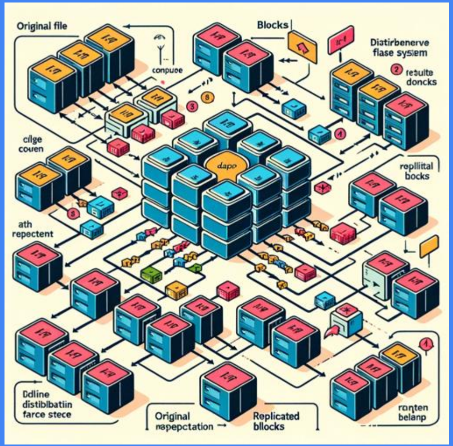
Hadoop distributed file system (HDFS)

- 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

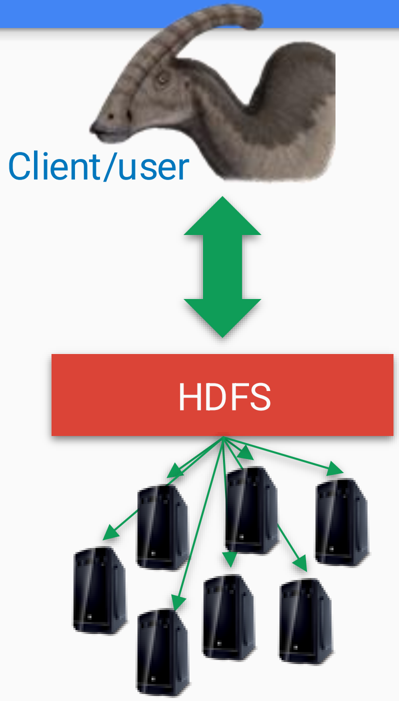
YARN
Resource manager



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



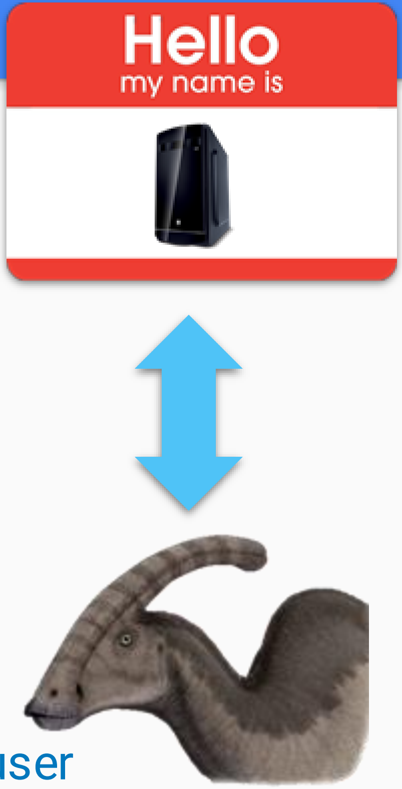
Two types of nodes (= computers) in HDFS

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



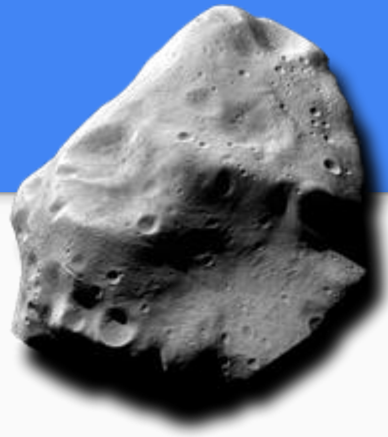
“Data” nodes (also where mappers and reducers are run)

- 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

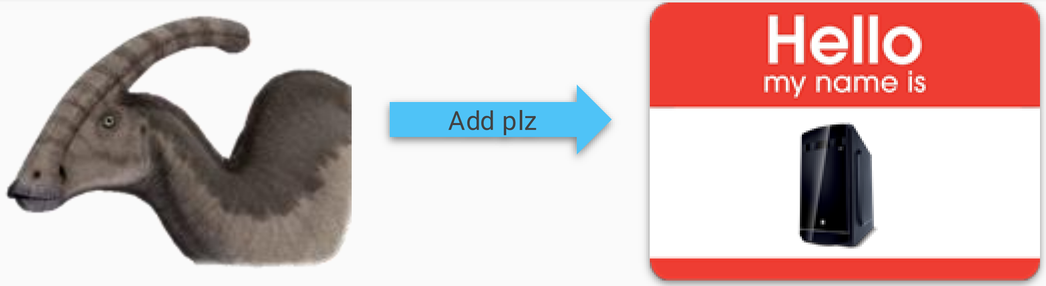
- **Generation stamp**: used to detect updates

Division of labor/responsibilities



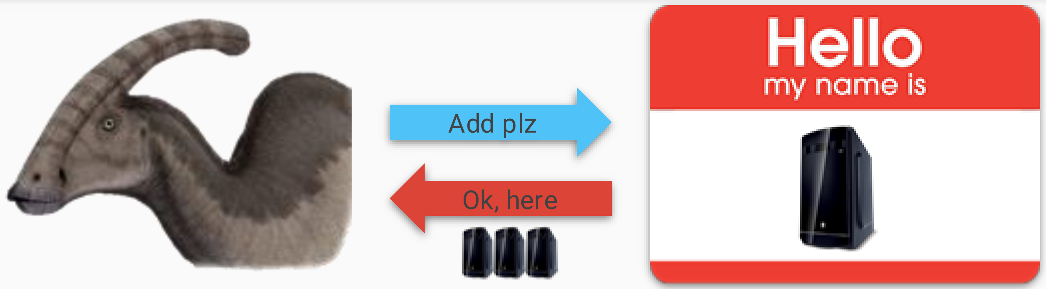
- Name nodes do not store **data**!
- Data nodes do not store file names/file structure!
- Name node failure is **catastrophic**
- Data node failure can be tolerated, up to a point
    - Depends on how many replications of each file you have

Example operation: writing to HDFS



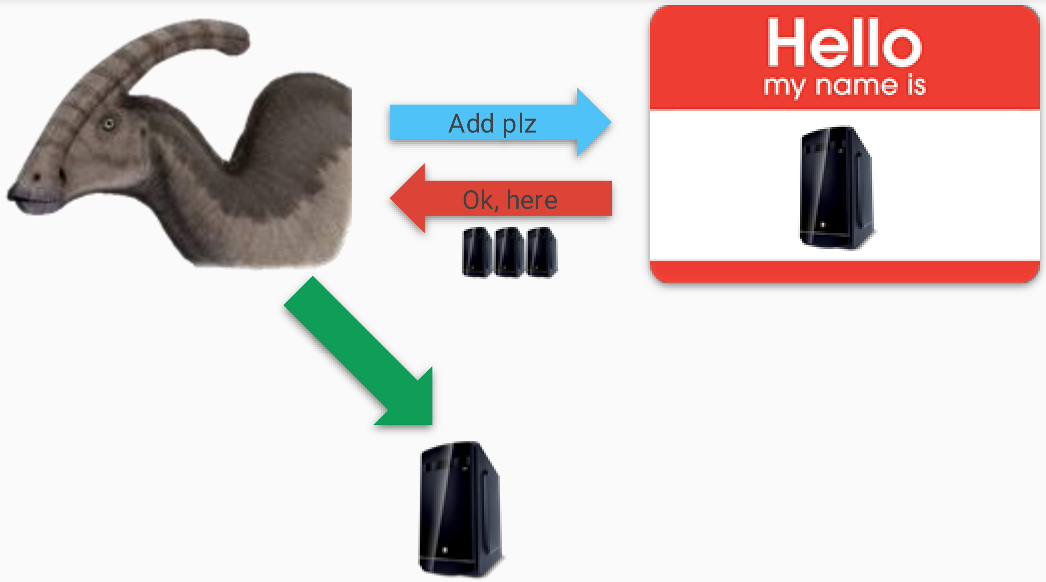
1. Client wants to add a block

Example operation: writing to HDFS



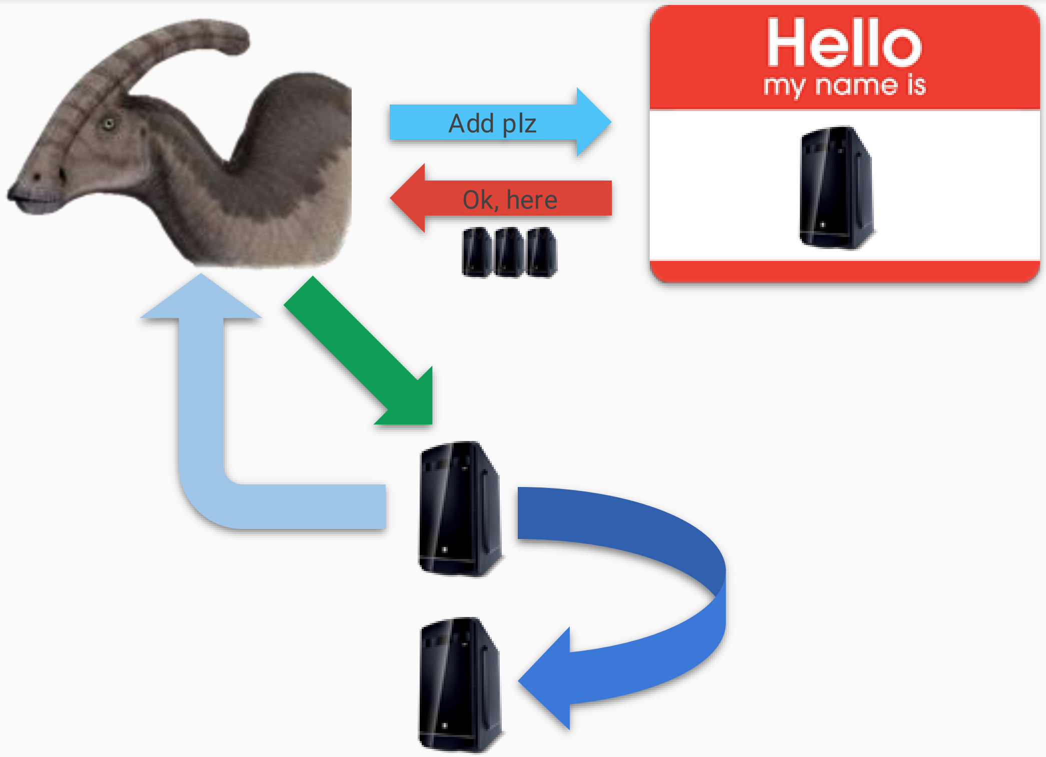
1. Client wants to add a block
   a. Name node responds with a list of data nodes

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

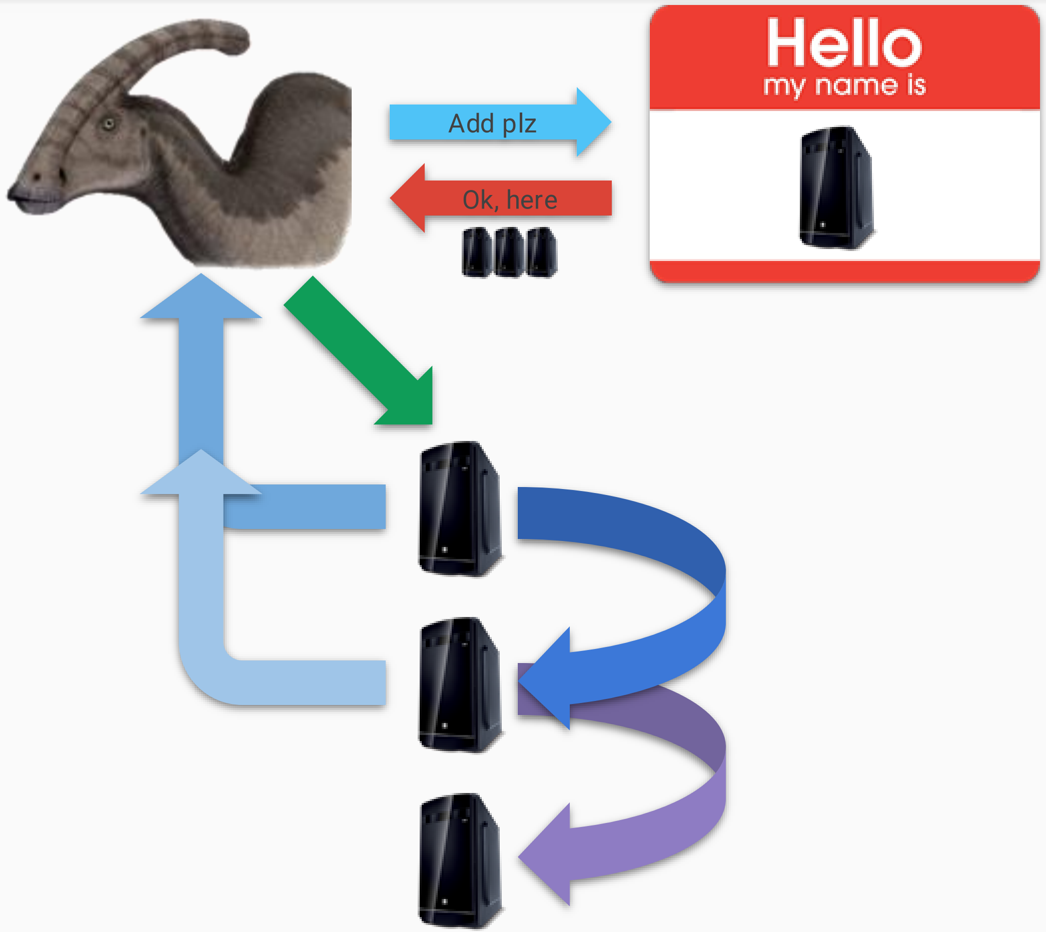


1. Client wants to add a block
   a. Name node responds with a list of data nodes

2. Client sends block to DN1

3. DN1 stores, acknowledges, and sends block to DN2

Example operation: writing to HDFS



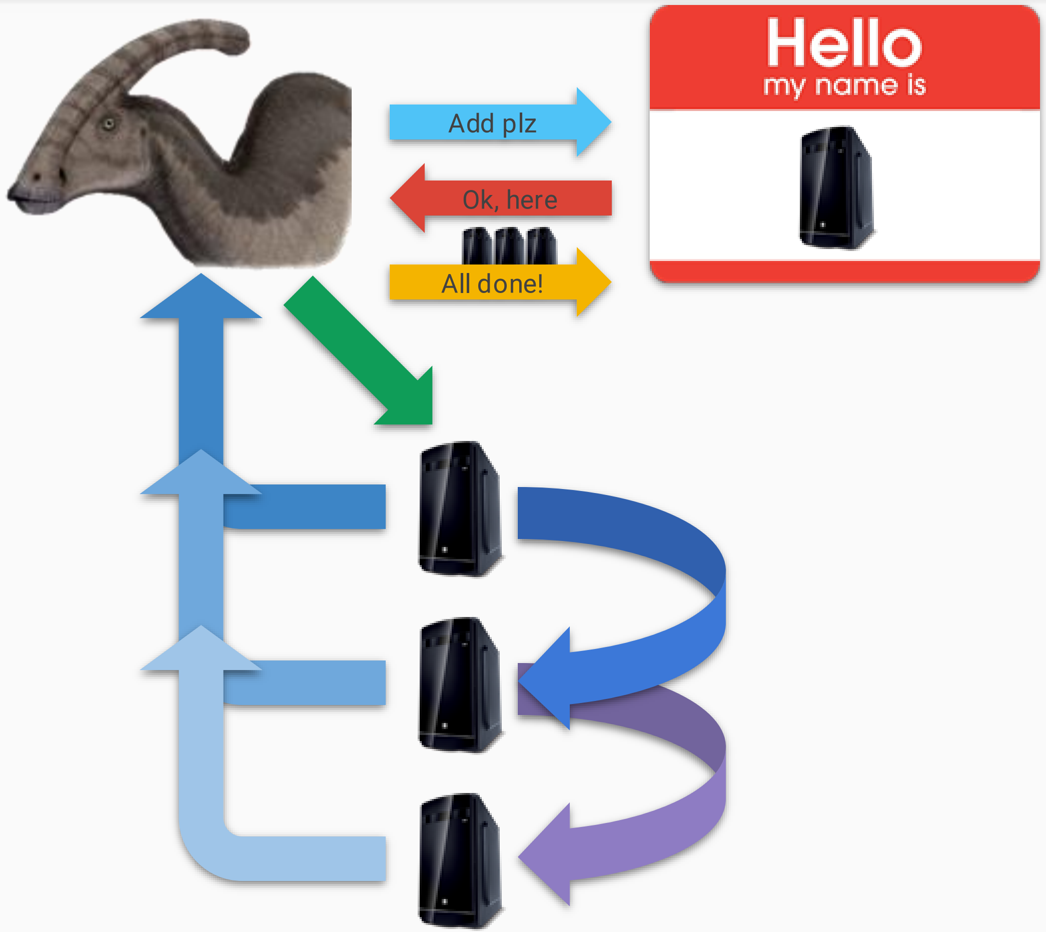
1. Client wants to add a block
   a. Name node responds with a list of data nodes

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

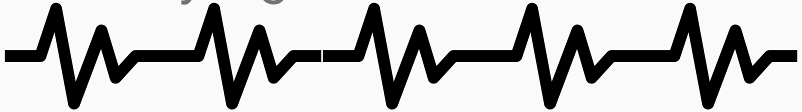
4. DN2 stores, acknowledges, and sends block to DN3

5. DN3 stores and acknowledges

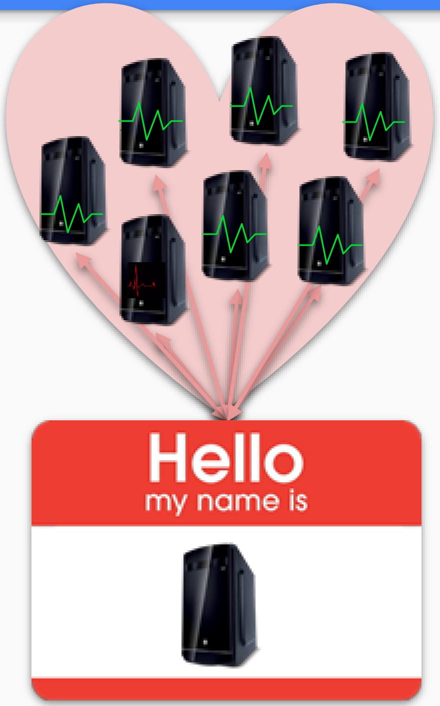
Add complete, close file, **alert name node**

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*

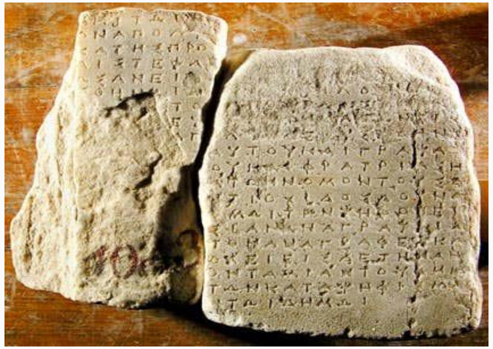


Recovering from name node failure: Checkpoints

- 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



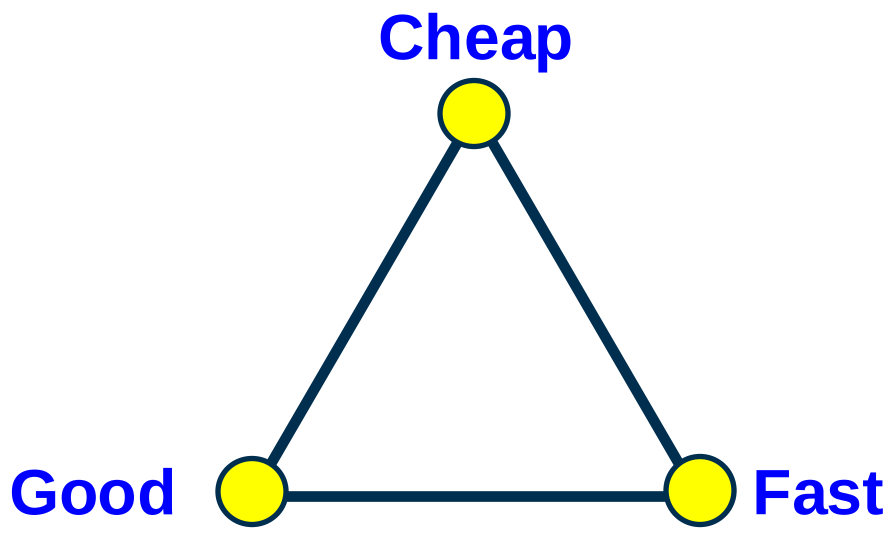
- Not all file modes are supported
  - Not all modes make sense in this limited context anyway
  - E.g., executable (neither desirable nor necessary for mapReduce)

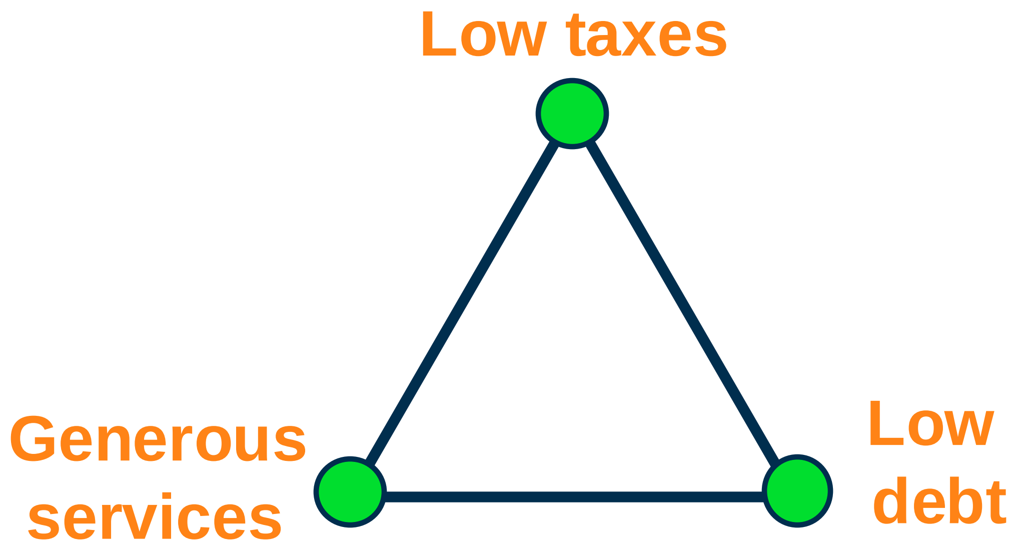
Why doesn’t HDFS work like the file system on my desktop?

- 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*



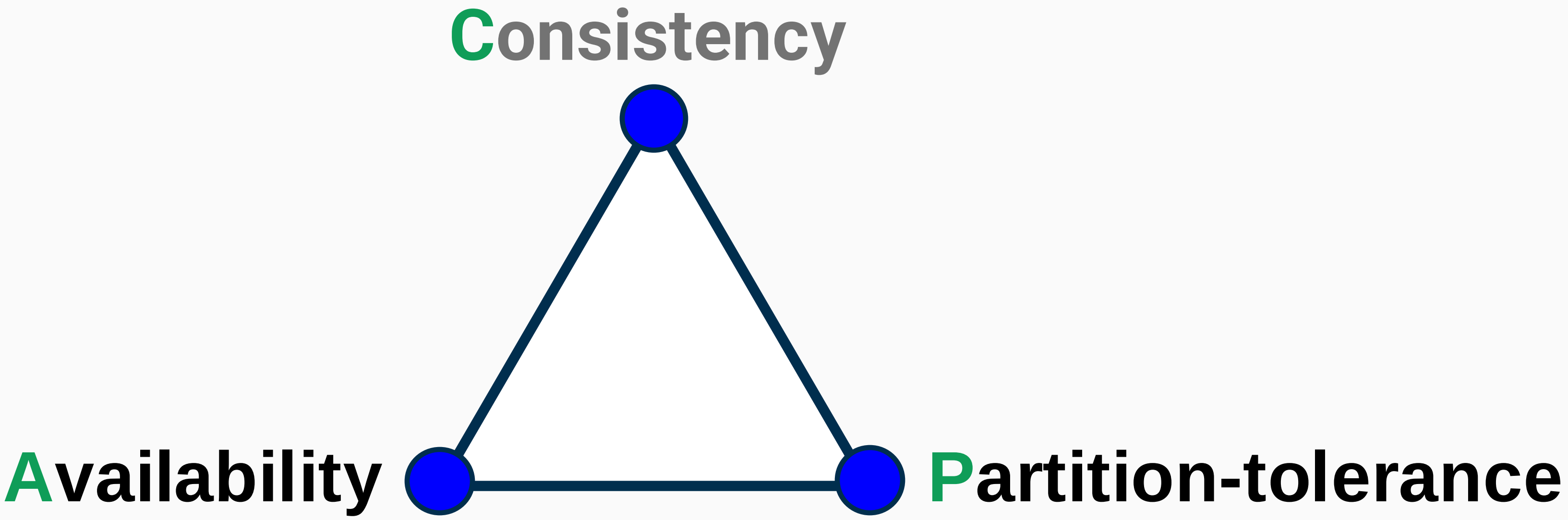


You can pick the nodes along ONE edge

*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]

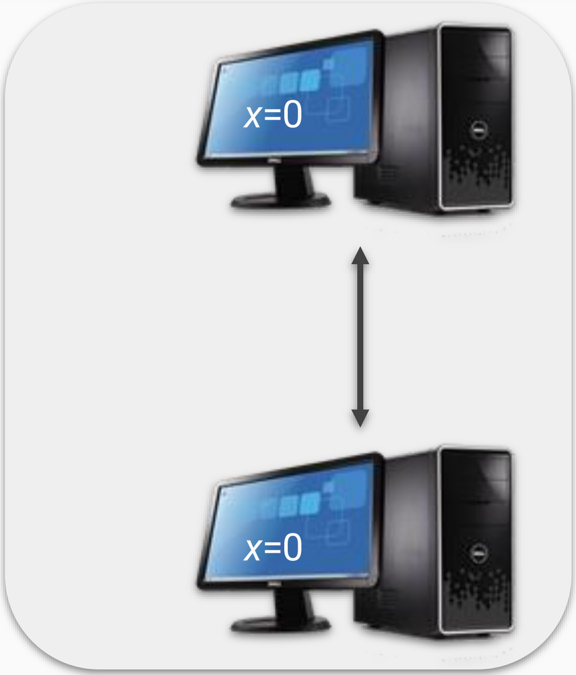
- **Consistency**:
  - Read always produces the most recent value
- **Availability**:
  - Requests cannot be ignored
- **Partition-tolerance**:
  - System maintains correctness during network failure

Pick which one you prefer

Gotta have this one

Why can't we have CAP? Assume that we could...

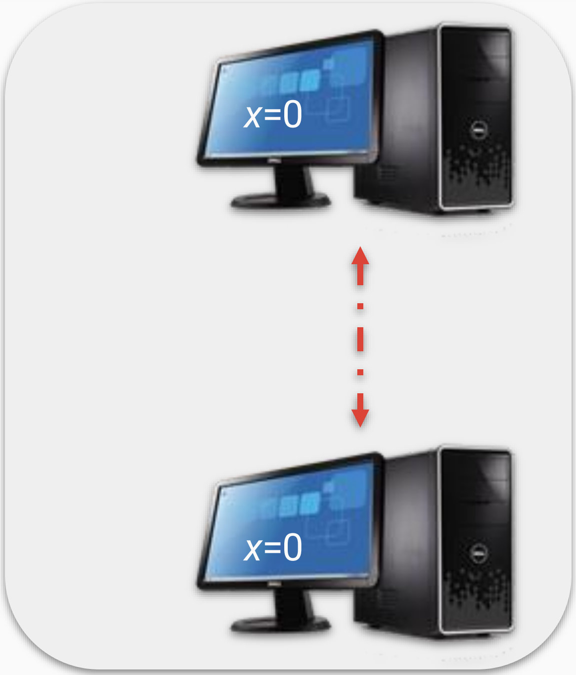
VelociCAPtor



1. M_1 and M_2 initialized with x=0

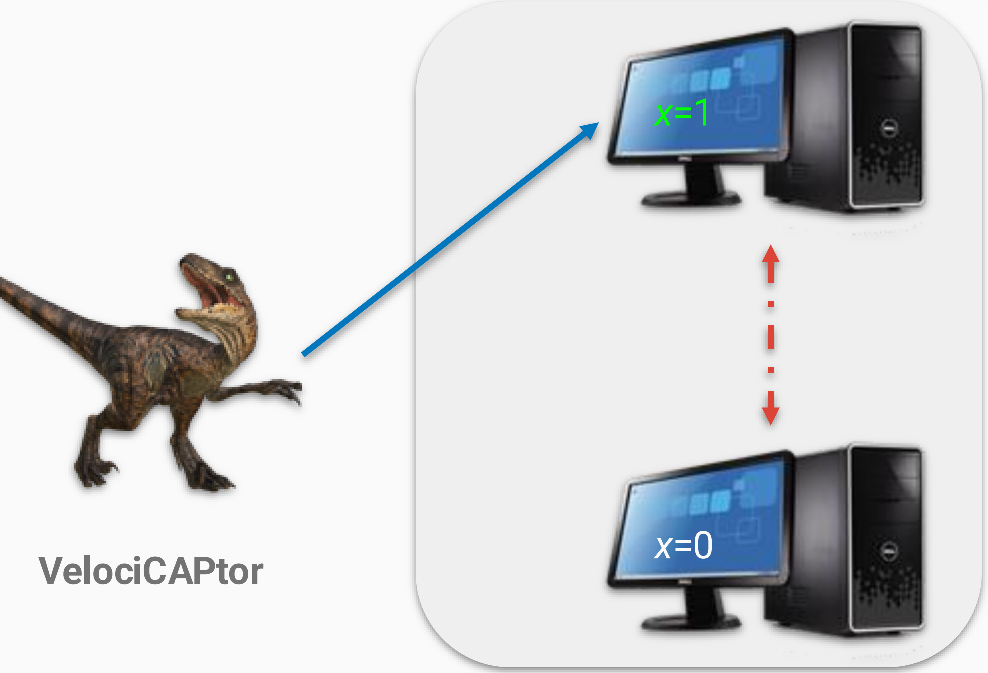
Why can't we have CAP? Assume that we could…

VelociCAPtor



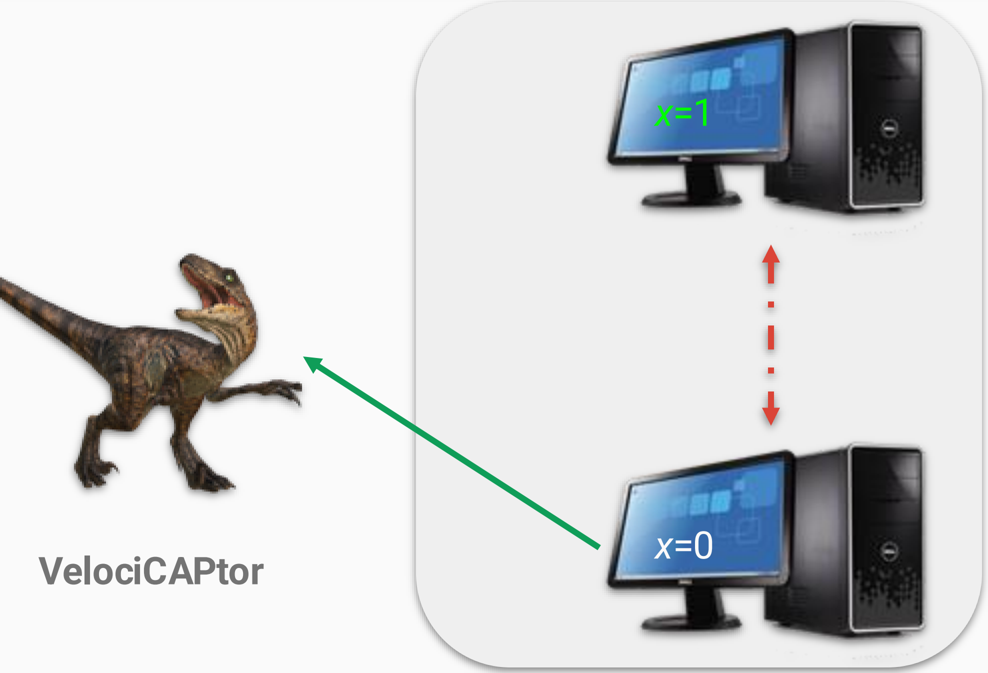
1. M_1 and M_2 initialized with x=0
2. Network fails

Why can't we have CAP? Assume that we could...



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!

Q&R

hadoop