Smallest font

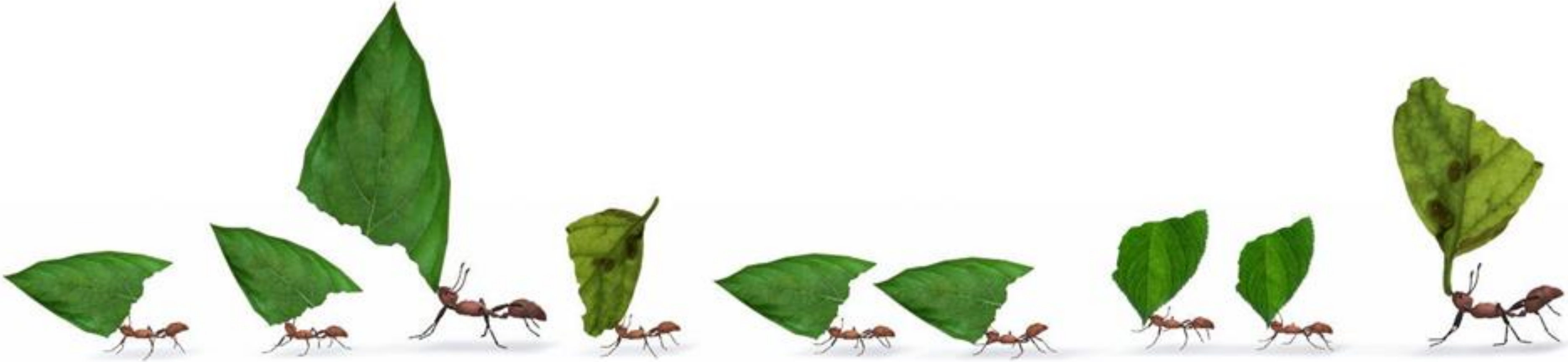
Please turn off and put away your cell phone

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

Week 03: Map Reduce

DS-GA 1004: Big Data

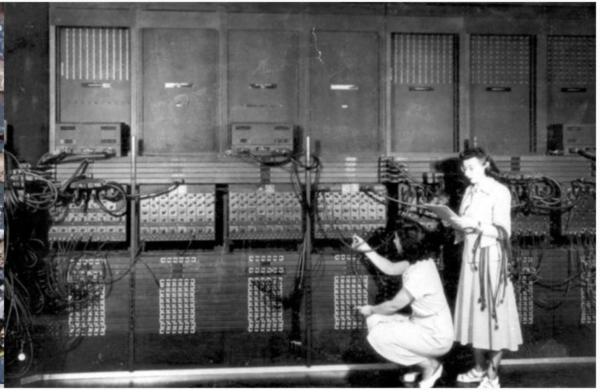
Announcements

- 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

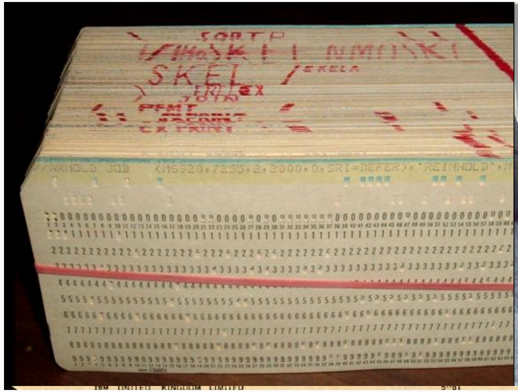
“Prehistory”
(not covered in this class)

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

1970s

1960s

Relational model (Codd, 1970)

Tables are good!

Structured Query Language (SQL)

1960s

File systems

Custom software for each application / query

1970s

Relational model (Codd, 1970)

Tables are good!

Structured Query Language (SQL)

1980s

RDBMS takes off

Databases for commodity computers

SQL “standardizes”

1960s

File systems

Custom software for each application / query

1970s

Relational model (Codd, 1970)

Tables are good!

Structured Query Language (SQL)

1980s

RDBMS takes off

Databases for commodity computers

SQL “standardizes”

1990s

The web happens

Distributed / parallel computing takes off

1960s

File systems

Custom software for each application / query

1970s

Relational model (Codd, 1970)

Tables are good!

Structured Query Language (SQL)

1980s

RDBMS takes off

Databases for commodity computers

SQL “standardizes”

1990s

The web happens

Distributed / parallel computing takes off

2000s

MapReduce

Simpler abstractions for distributed computation and storage

Tables are bad!

1960s

File systems

Custom software for each application / query

1970s

Relational model (Codd, 1970)

Tables are good!

Structured Query Language (SQL)

1980s

RDBMS takes off

Databases for commodity computers

SQL “standardizes”

1990s

The web happens

Distributed / parallel computing takes off

2000s

MapReduce

Simpler abstractions for distributed computation and storage

Tables are bad!

2010s

THE CLOUD ERA

Flexible cluster management

High-level languages

Focus on interactivity

Tables are good if you call them ✨ data frames✨

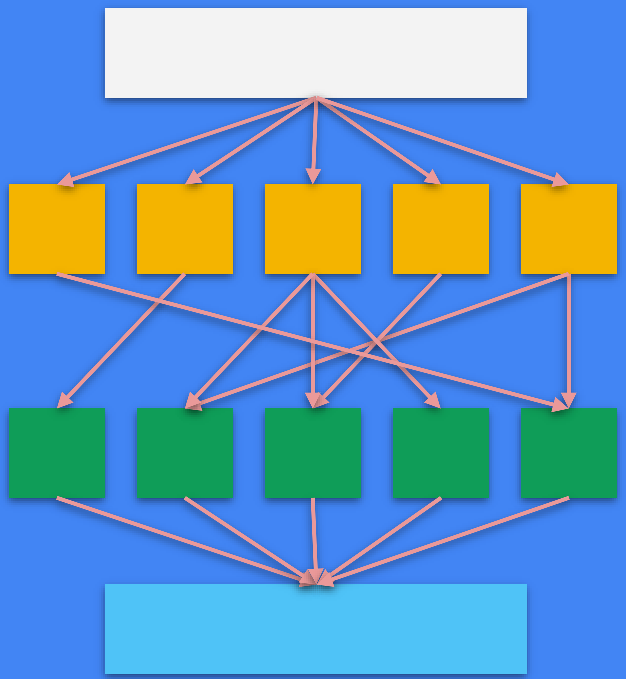
Reminder: Last time - RDBMS

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

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

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

This week

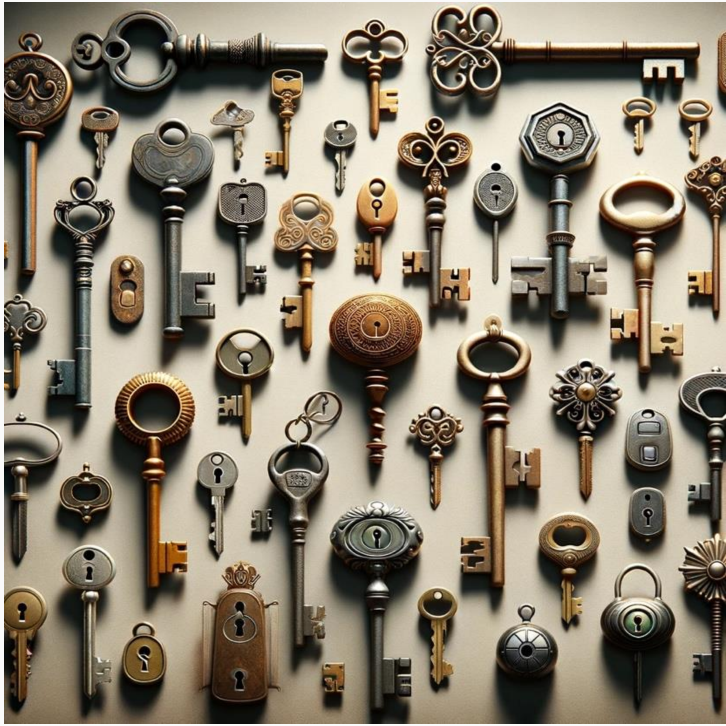


1. Resolving confusions / doubts / struggles from AAA
2. Introduction to Map-Reduce (Dean & Ghemawat, 2008)
3. Assessment of Map-Reduce (DeWitt & Stonebreaker, 2008)

Confusion, Doubt, & Struggle: **Keys**

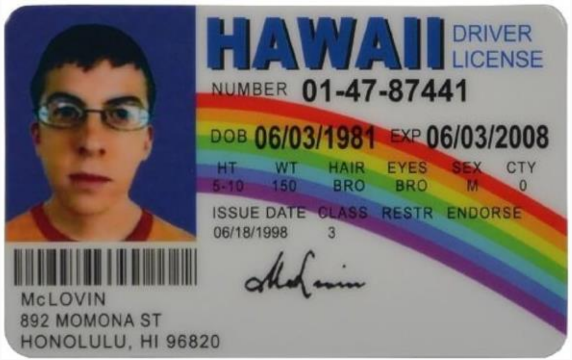
"Keys" are a central concept in many CS applications

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

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

(Primary) key

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

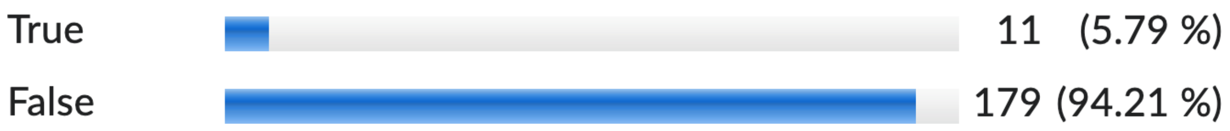
Confusion, Doubt, & Struggle: Index

- 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);

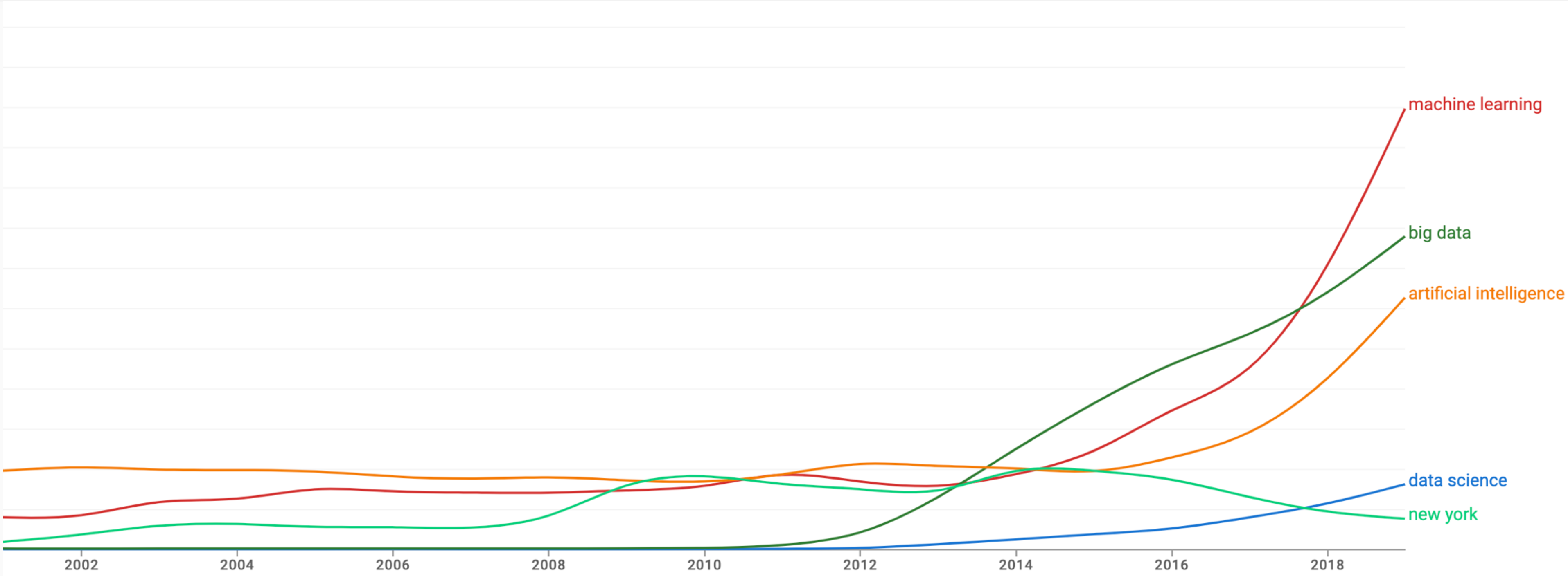
NOW

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]

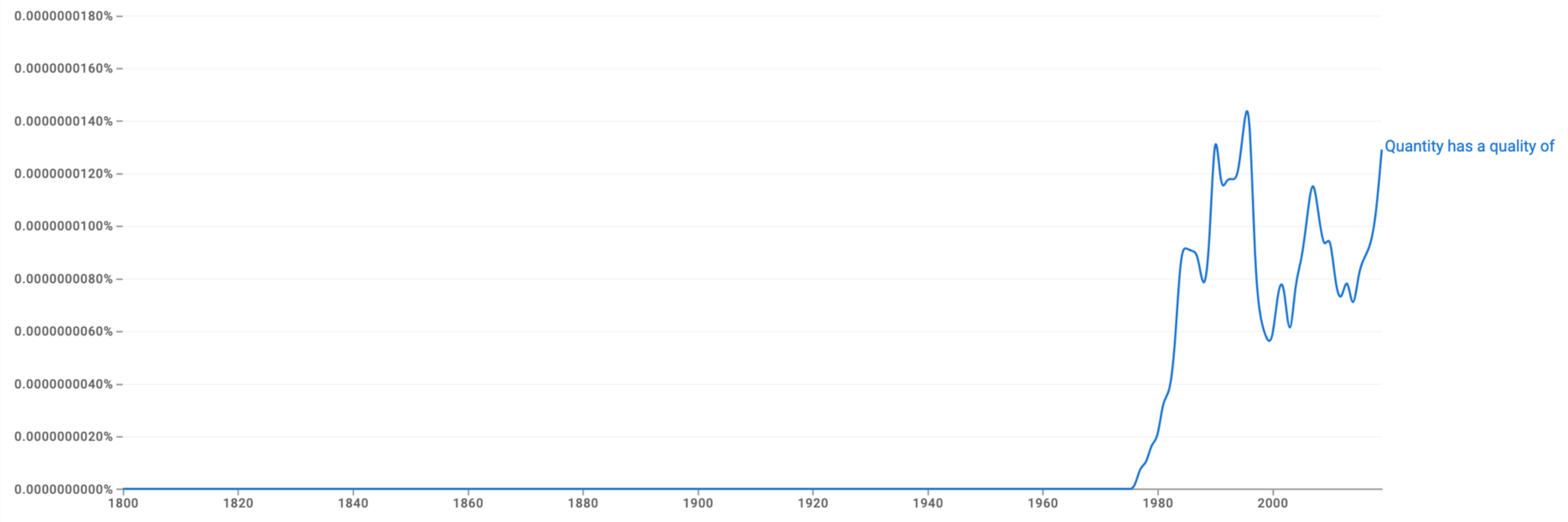


Challenge: A tremendous **collection** of **documents** needs to be searched to do something like this

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

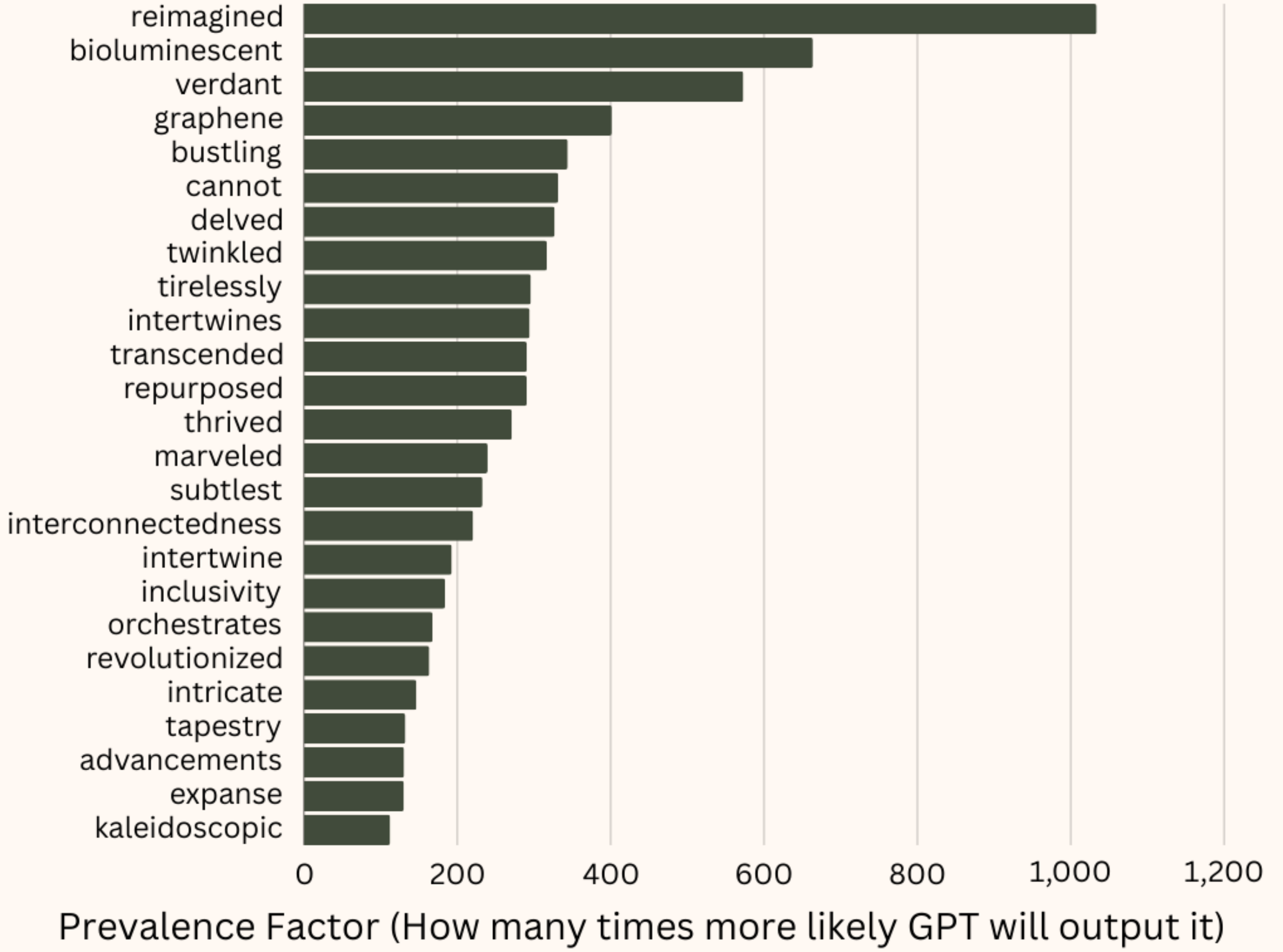
Why Map Reduce was created

- 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** $\rightarrow$ **documents**

- On a single machine, this process takes $\boldsymbol{\Omega(N)}$ time

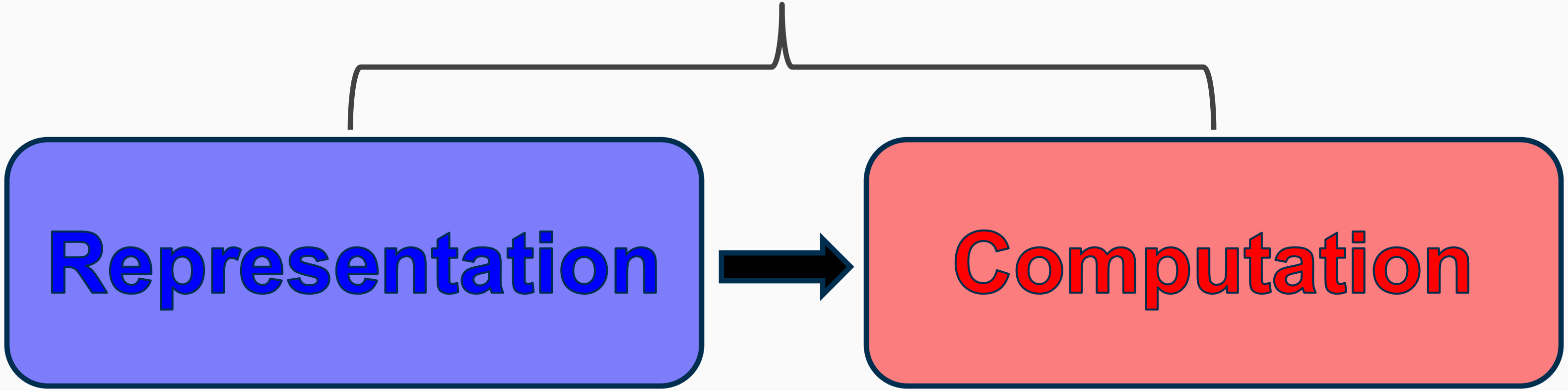
- **Observation**: this problem is (almost) embarrassingly **parallel**
  - Whether any word appears in a document is **independent** of other documents
  - We should be able to process documents independently and **combine the results**

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

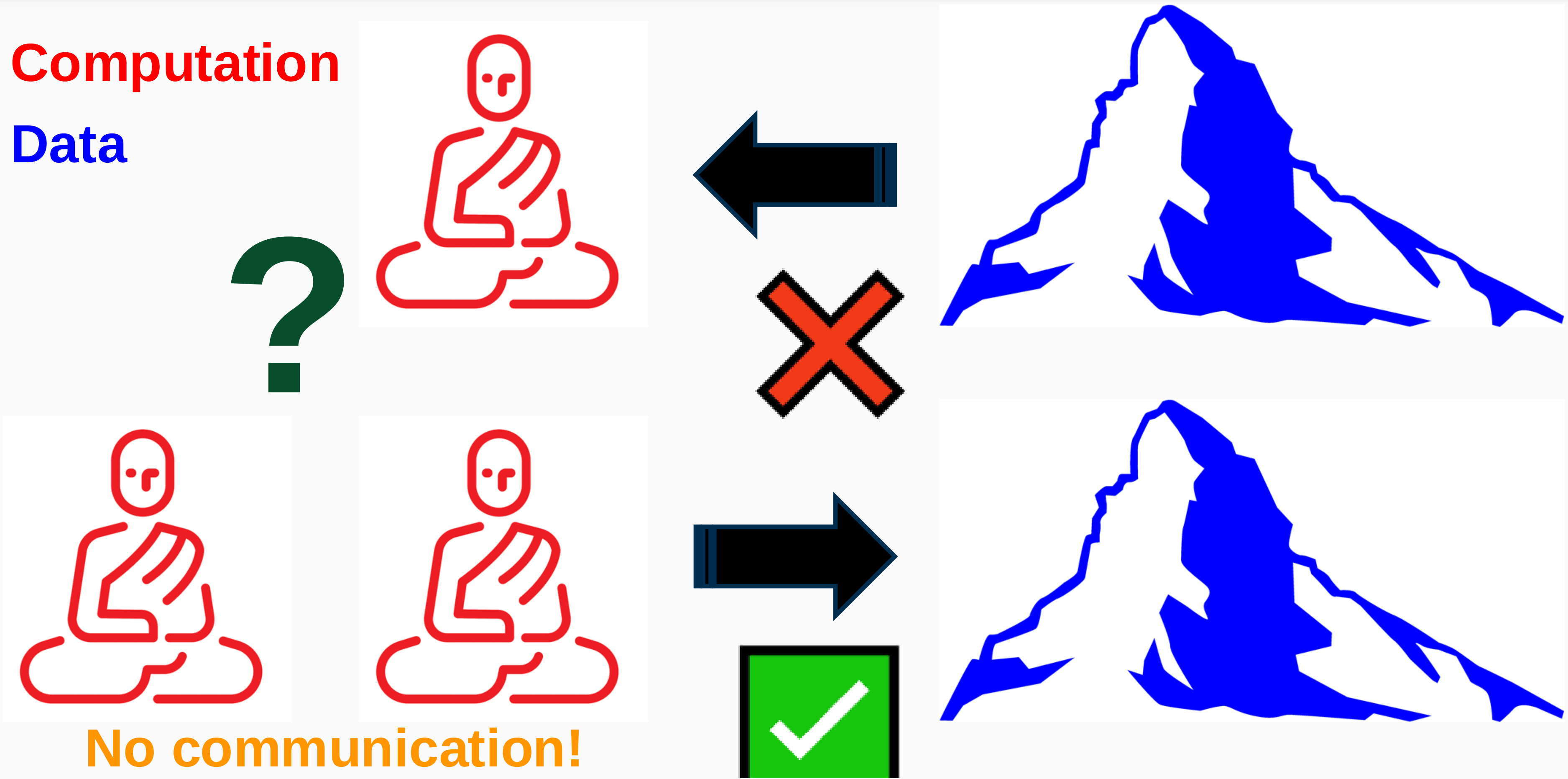
Data Processing



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

Why do such a thing?

“Data Locality”



Power through restrictions and constraints

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

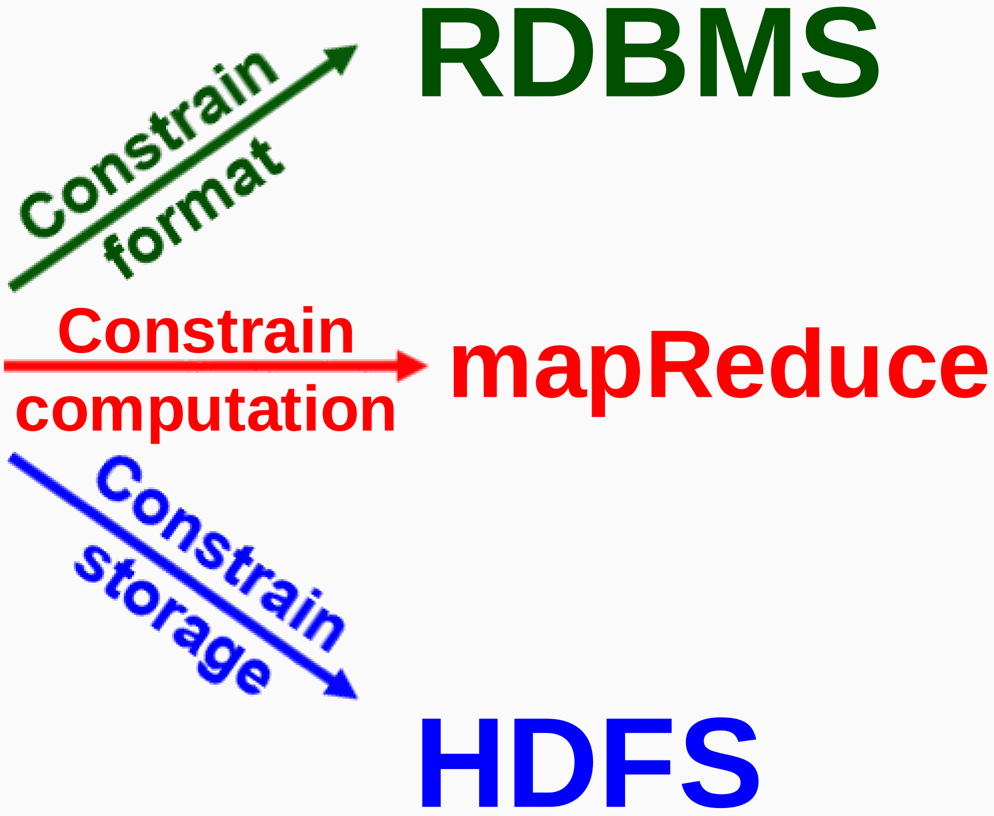
File systems

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 $[x_1, x_2, \ldots, x_n]$) $\rightarrow [f(x_1), f(x_2), \ldots f(x_n)]$
  - **map** : function, list $\rightarrow$ list
  - f is applied element-wise
  - This allows for parallelism, if list elements are independent
- **reduce**(function g, values $[x_1, x_2, \ldots, x_n]$) $\rightarrow g(x_1,$ **reduce**$(g, [x_2, \ldots, x_n]))$
  - **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
   a. Distribute data to mappers
   b. Generate intermediate results (*key*, *value*)

2. Sort / shuffle phase
   a. Assign intermediate results to reducers (by *key*)
   b. Move data from mappers to reducers

3. Reduce phase
   a. Execute reducers and collect output

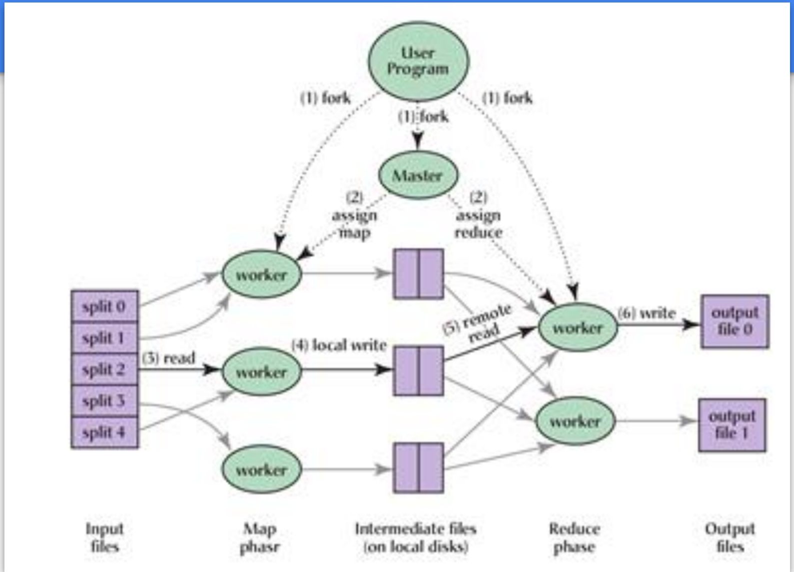
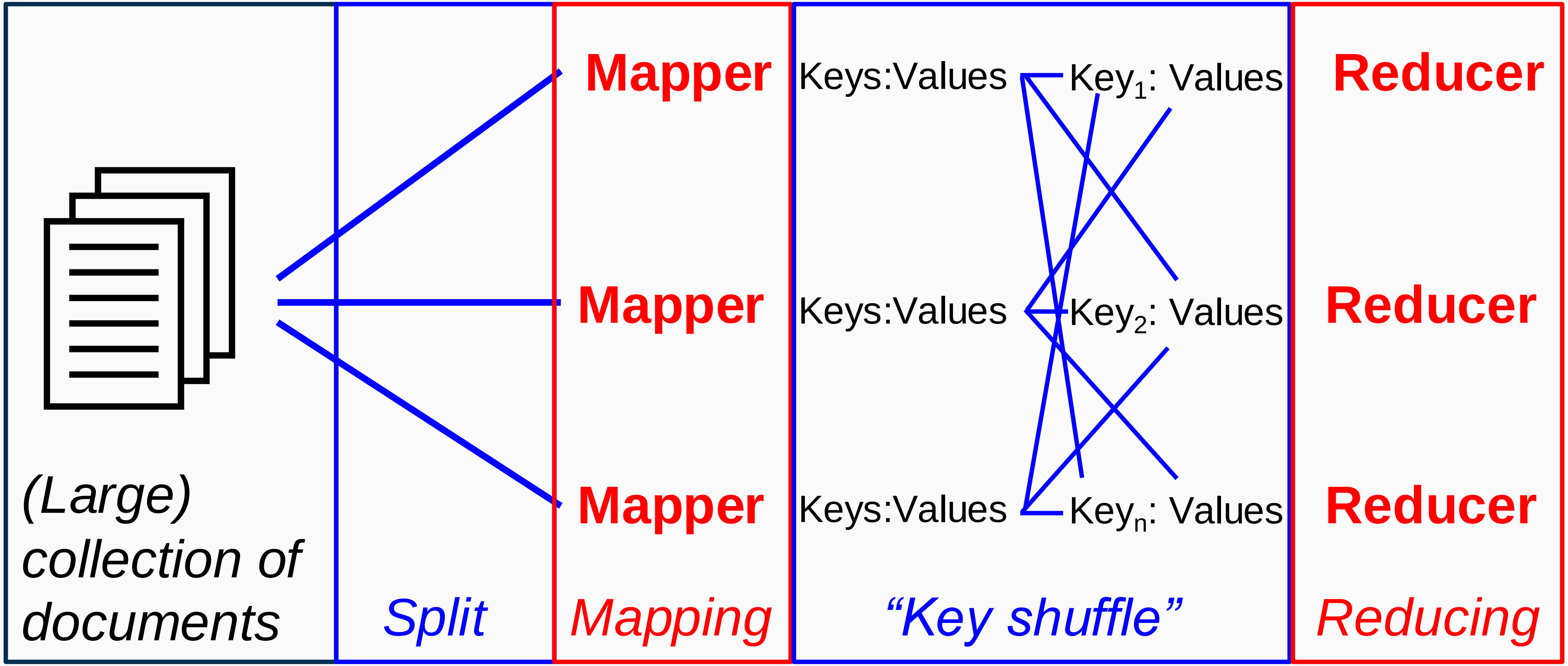


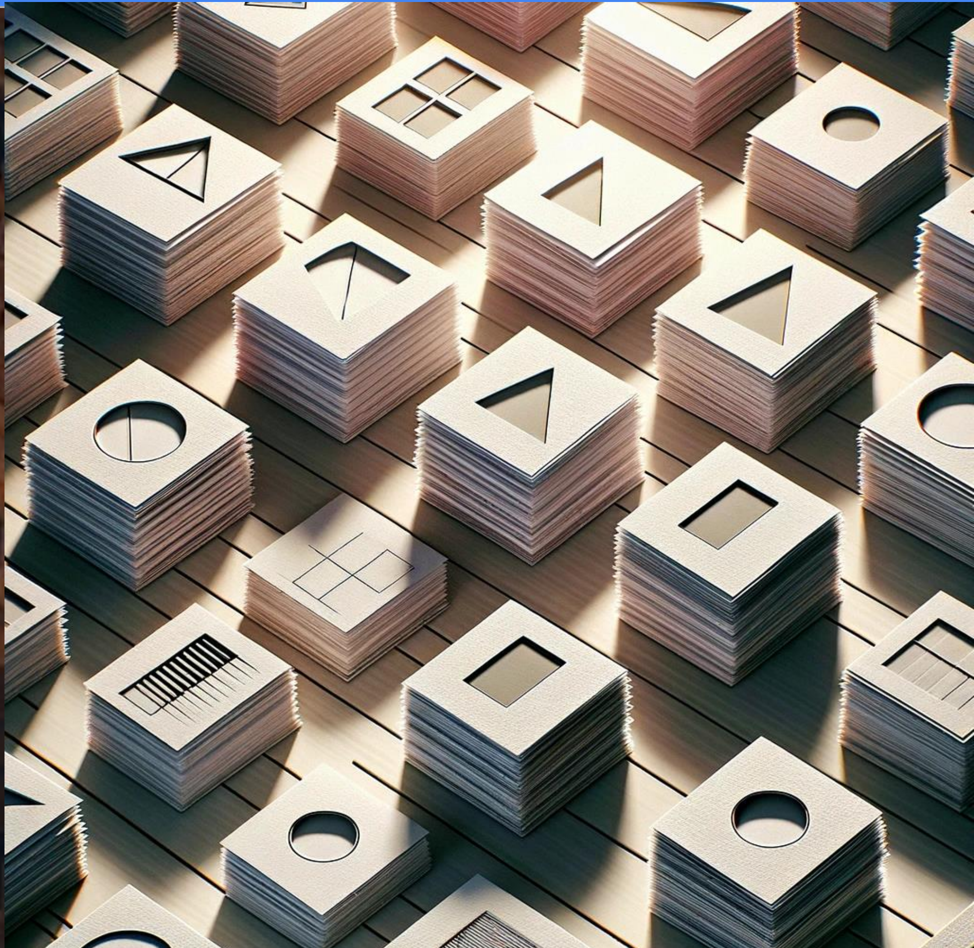
Figure from Dean & Ghemawat, 2008

The MapReduce algorithm (essential schematic)

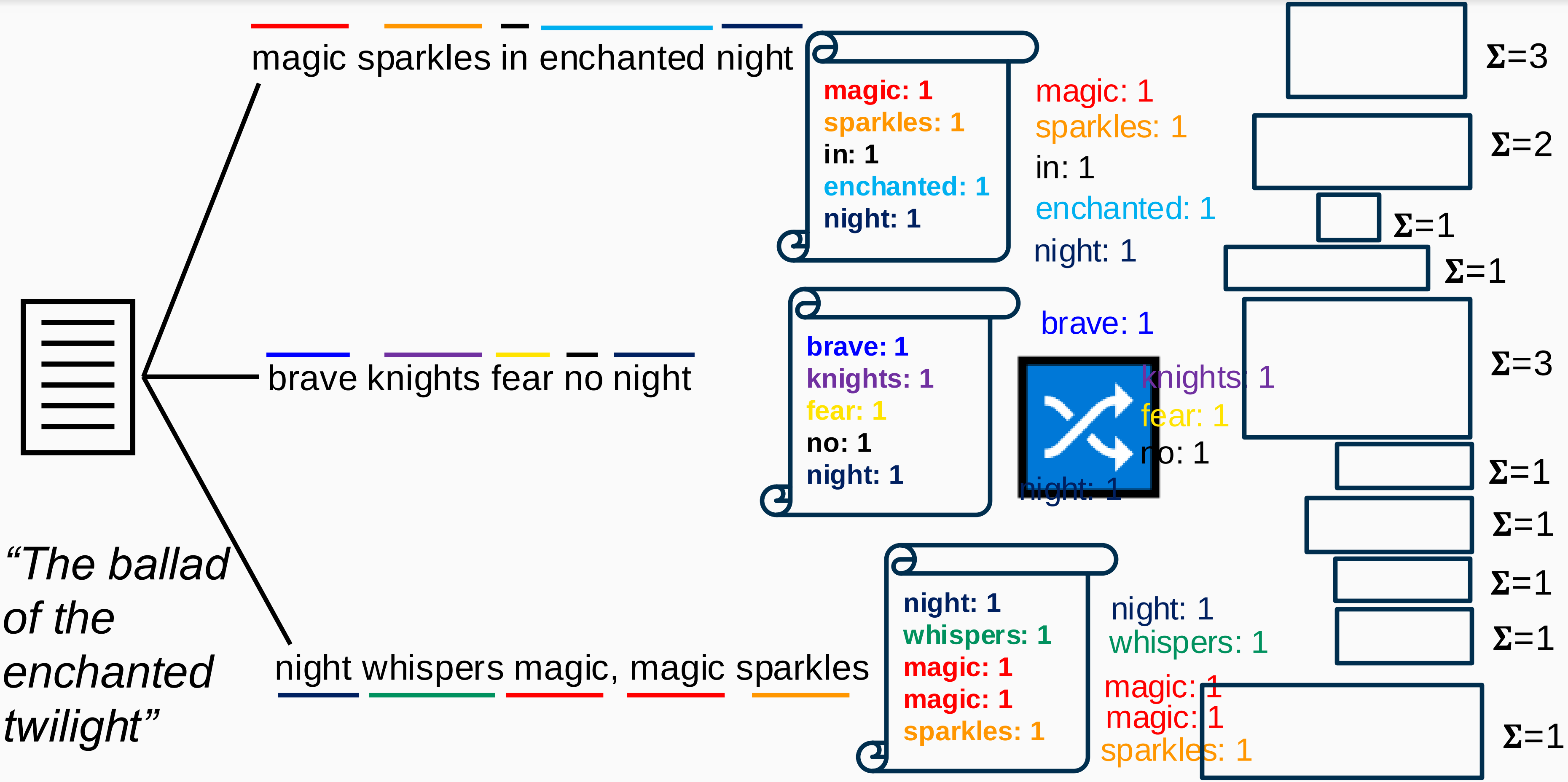
Little movement of **data** and no **communication** between **computational** units
= massively **scalable** due to **parallelization** → “finish in reasonable time”



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: “**D**ivide **E**t **I**mpera” (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:

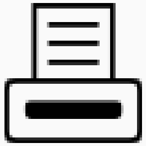
Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c))$ – order of operations matters not

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

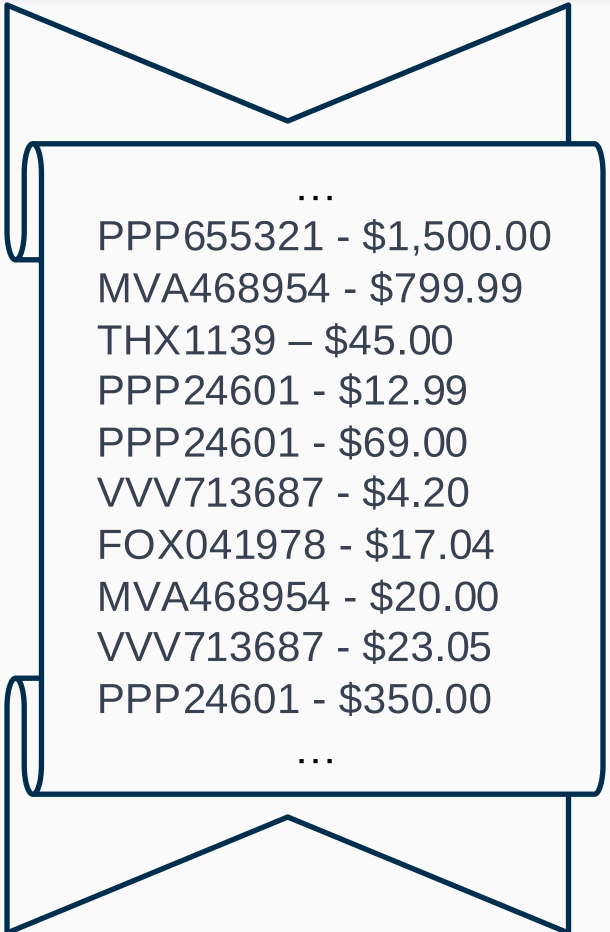
Determinism: Same inputs yield same outputs, without side effects.

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!

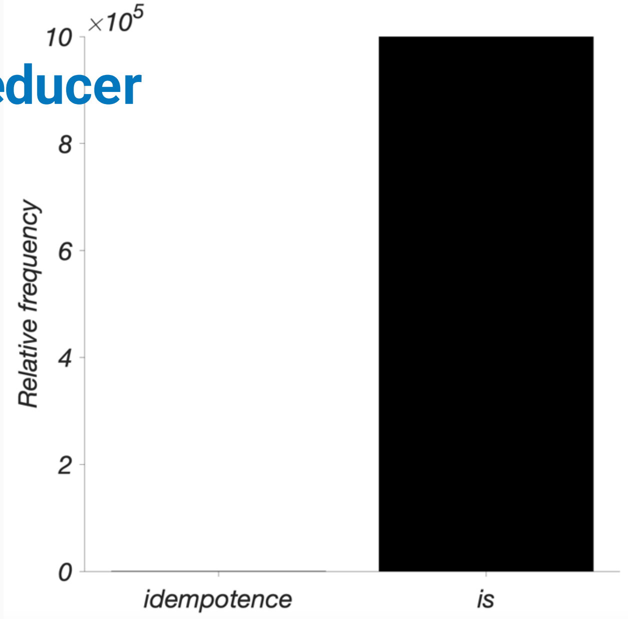
Key $\rightarrow$ reducer assignment

- **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 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

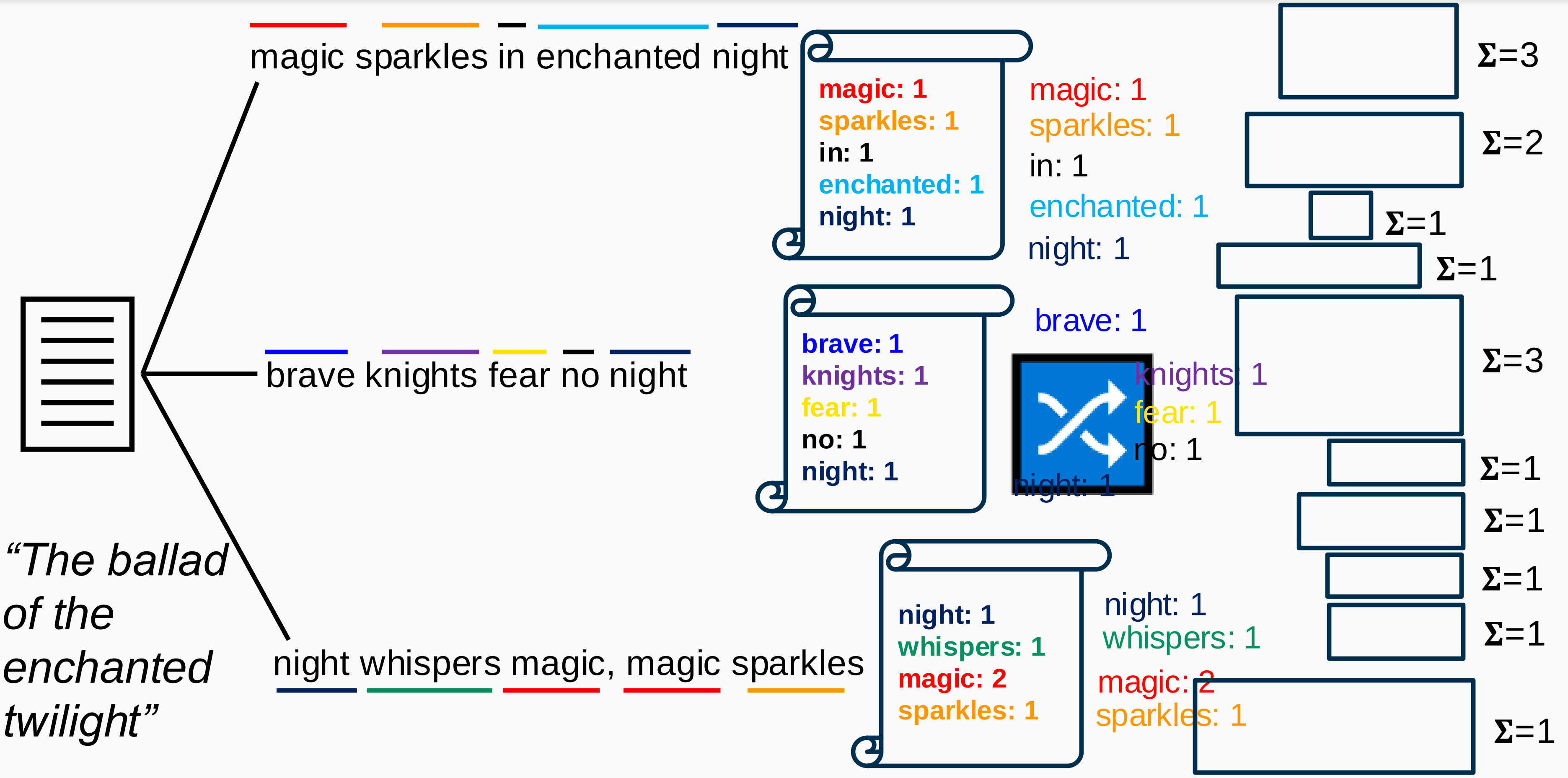
This works because summation is **commutative** and **associative:**

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 2nd 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