Overview

1. Data Storage, Representation
2. Small, Medium, or Big Data?
3. Big Data Algorithms
4. MapReduce

Data Storage

- Everything is just 0s and 1s: bits
- Everything is measured in terms of bytes, which is technically 8 bits, but we can often just treat it as 10^1 bits for back of the envelope calculations

Data Storage

- Small Image `~7 KB = 1*10^3  bytes`
- MP3 `~5 MB = 5*10^6  bytes`
- Human Genome `~3 GB = 3*10^9  bytes`
- Memory in laptop `~4 GB = 4*10^9  bytes`
- Desktop drive `~2 TB = 2*10^12 bytes`
- LHC per year `~15PB =15*10^15 bytes`

Data Storage

- How much space for a number?
  - **Integer?** 4 bytes, 8 bytes, more?

    Max Int64: ≈$9 * 10^{18}$
  - **Floating point?** “double precision” 8 bytes

    Max Float64: ≈1.8×10^{308}

Data Storage

- “Tiny” 1 MB $\approx 10^6$ bytes $\approx 10^5$ numbers
- “Small” 1 GB $\approx 10^9$ bytes $\approx 10^8$ numbers
- “Large” 1 TB $\approx 10^{12}$ bytes $\approx 10^{11}$ numbers
- “Big” 1 PB $\approx 10^{15}$ bytes $\approx 10^{14}$ numbers

Strings

- Data often isn't in "binary" form - frequently represented as a **string** (possibly compressed)
- Past: **"ASCII"** et. al.
  - Usually **1 byte** per character
  - Non-English 2 bytes per character, non-compatible
- New standard: Unicode, esp. **"UTF8"**
  - English characters **1 byte** (8 bits)
  - Non-English characters 2-4 bytes

Representation Matters - trips.csv

4 bytes each (at most!): “id”, “duration”, “start_station”, “end_station”, “bike_nr”, “subscription_type”, ”zip_code”, ”birth_date”, ”gender”

8 bytes each: “start_date”, “end_date”

```
$ du -h trips.csv                           55 MB
$ wc -l trips.csv                           552074
$ echo "552074*(4*9+8*2)" | bc              29 MB
$ gzip trips.csv; du -h trips.csv.gz        10 MB
```

Just How Much Trouble Am I In?

- **"Big Data"** is overused to the point that talking about how overused it is cliche
- Let's establish a taxonomy!
- Discuss what sort of things you need in each situation

Small Data (< 10 GB)

- If it can fit in memory of your laptop...
- If it comes as an Excel file…
- Then no special tricks typically needed
- **“Use a bigger computer”** is usually the **easiest solution** (if its an option)

Small Data Deceptions

- **Compression** can hide issues, e.g. compressed file 2GB -> in memory 10 GB
- **Poor representation/extra data** can look like "medium data"

- e.g. 10 GB of Hubway trip data but only need start station, end station

Real Example

- From Hubway Data Visualization Challenge:
    - “Station status data, with information about available bikes and empty docks per station and minute back to August 2011 (30 million records), is now available: download **190MB CSV (tar.gz)**”
- Uncompressed its 1.4 GB
- Might be even bigger when read into R/Julia
- Still loadable into memory all at once though.

Medium Data (10GB - 1TB)

- Too big to load into memory all at once
- **Filter the data** or **partially load the data**
  - Row-by-row, accumulate/store only what you need
  - Manually or use packages to load in chunks
  - Cache temporary results
- Operations on raw data
  are no longer obvious,
  e.g. linear regression

Iterating Through Data

- Live coding session
- Follow along with `iterate_script.jl` if you want

Assignment 1: Iterating and Memory

1. Open Julia
2. Load `trips.csv` all at once
3. Close Julia, then open it again.
4. Use `eachline()` to loop over the file, calculating average duration (instead of loading it all first)

Check memory usage after each step!

- **Windows:** Ctrl+Shift+Esc
- **OSX**: Activity Monitor in Utilities
- **Linux**: top

Prompt file: `assmt1_iterate.jl`

Iterating Through Data

- Initial 42
- After **`readlines`** 161
- Re-open, initial 42
- After running **`eachline`** 65

Working with Partially Loaded Data

- Reading/writing to disk is slow
  - Expect things to be a lot slower
- Many packages for R, e.g.

  `ff`, `bigmemory`,

  `biganalytics`, **`biglm`**,

Demonstration of `ff, biglm`

- See `ffbiglm_script.r`

Relevant Documentation

ff: http://cran.r-project.org/web/packages/ff/index.html

biglm: http://cran.r-project.org/web/packages/biglm/index.html

Another option: bigmemory family

- http://www.bigmemory.org/
- Limitation: matrices, not dataframes, so must be uniform type across data (except header)
- biganalytics, bigalgebra

Other Medium Data Ideas

- Do you need all your data?
- Can you split your analysis into multiple stages?
- If your intermediate results don't fit in memory, write them to disk, then read them back in again

Assignment 2

Write a program in Julia (or whatever) that

1. Takes **trips.csv** and extracts the start_station and end_station for all trips with a duration between 60 and 3600, and saves them to **reduced.csv** as you go
2. Load **reduced.csv** line-by-line and determine which **pair of stations had the fewest trips (but > 0) & most trips**.

Hints: not all station IDs are used. Maybe record a list of station ids as you load **reduced.csv**? You can store the matrix of start-end trips in a matrix or dictionary-of-dictionaries (extra: can you be even smarter?). *Prompt:* `assmt2_twofie.jl`

Big Data - 10 TB+

- Maybe doesn’t fit on a single computer
- Multiple machines, storage arrays...
- Approaches: MapReduce, sampling...
- General approach:

  “Bring the computation to the data”

- IO-bound, not compute-bound

Big Data Examples

- **Google** has, say, O(10^12) pages in index
- Store just 1 KB (10^3 bytes) about each
- Require 1 PB (10^15 bytes) of information!
- They store a **lot** more!
- Does anyone one question you want to answer about the index need it all?

Big Data Examples

- Facebook has O(10^9) users.
- Say each user has a ID number (8 bytes)...
- … and approximately 1000 friends.
- If storing the “friend graph” as adjacency list then you’ll need 10TB just for that.
- Pictures are big: 1MB*10^9 = 1PB of space just for profile+cover pics!

Reservoir Sampling and Streaming

- If you can’t analyze whole data set, sample / take a subset
- Sometimes as easy as just taking every *n*th row, but...
- … need to make sure you are sampling correctly
- How to do things in linear time/single pass?

Algorithm 1: Uniform Sampling

- Say we have a list of items of length *n* and want to pick 1 item at random (e.g. *1/n* each)
- **Not good**: for big file need to read through the file once to determine *n*, pick *i*, then run through again until we reach *i*
- **Even worse**: data is arriving as a stream, need to save stream to disk so can re-read!

Algorithm 1: Uniform Sampling

- Challenge: sample in one pass. Class of algorithms is called “reservoir sampling”
- **Solution**:

```
Define x to be selected item
Set x is first item, i = 1
While not at end of file/stream
    i = i + 1
    Replace x with cur. item with prob. 1/i
```

Assignment 3: Don’t trust me, code!

Write a Julia function that takes an argument *n*

1. Run this algorithm 10,000 times over 1...n, record which one was selected each time
2. Prints out the number of times each i was selected as a percentage

Can you think of how to modify to take every possible subset of size m with equal probability? *(no prompt)*

Algorithm 2: Find Most Frequent

- I have a stream (of length n, unknown) where I know 1 item appears more than n/2 times
- How do you find it when...
  - Only running through the stream once
  - There are a large number of possible values
  - You have extremely limited storage (in fact, you can store only two numbers!)
- Naive approach ruled out by last bullet point

Algorithm 2: Find Most Frequent

- “Streaming algorithms”, “sketch” of a stream
- Algorithm:

```
Store “best” item B, and a count C
B = first item in stream, C = 1
While stream not finished
    If cur. item != B, and C = 0, set B = cur.item, C=1
    If cur. item != B, and C >= 1, decrease C by 1
    If cur. item == B, increase C by 1
Return B
```

See sketch_script.jl for code

Algorithm 2: Find Most Frequent

```
Stream: [A A B C A D D ]
1:A -> Selected = A, Count = 1
2:A -> Selected = A, Count = 2
3:B -> Selected = A, Count = 1
4:C -> Selected = C, Count = 1
5:A -> Selected = A, Count = 1
6:D -> Selected = D, Count = 1
7:D -> Selected = D, Count = 2
```

References

- Reservoir Sampling analysis:
  - Vitter, "Random Sampling with a Reservoir", http://www.cs.umd.edu/~samir/498/vitter.pdf
- Sampling:
  - Charikar et al, "Finding Frequent Items in Data Streams", http://www.mathcs.emory.edu/~cheung/papers/StreamDB/Frequency-count/FrequentStream.pdf
  - Applied to Google searches - one-pass-only essential

“MapReduce”

- You may have heard of something called:
  - MapReduce
  - Hadoop
- If you haven’t, don’t worry.
- We mentioned it earlier as a way to approach truly Big Data problems.

What is “MapReduce”?

- **MapReduce** is a **programming model**, or **methodology**, for building software to operate on big data.
- An ***implementation*** (also called MapReduce) of these ideas at **Google** popularized the concept, and is still in use.
- Google’s MapReduce is not (directly) available, so “MapReduce” usually refers to the ***concept***

What is “Hadoop”?

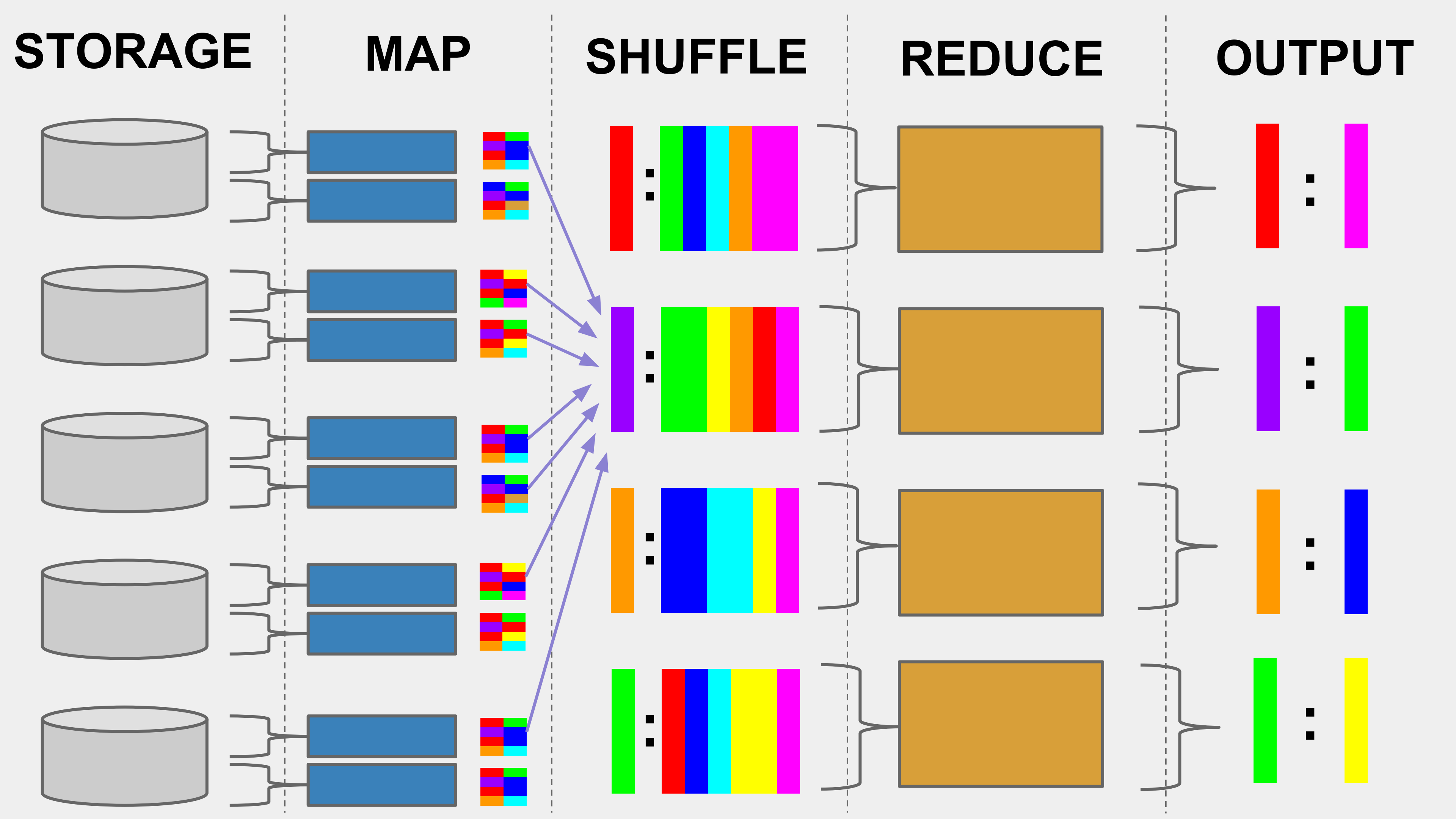
- Apache Hadoop is the most popular (I think) implementation of the MapReduce methodology.
- It can refer to the software that facilitates the MapReduce computation (more on this soon)
- Distributed file storage system **HDFS** for working with “big data” - H is for Hadoop

What We Won’t Do Today

- We are **not** going to discuss particulars of **Hadoop**
- Many options, and if you go work at a **Twitter**/**Facebook**/**Google**, etc. you may use a custom solution anyway.

The MapReduce Methodology

- **MAP** - For every item, apply the MAP function and output 0, 1, …, pairs of form `<key : value>`
- **SHUFFLE** - Done internally by framework, groups and sorts by key, outputs `<key : val1, val2, val3, ...>`
- **REDUCE** - For every key, output value(s)

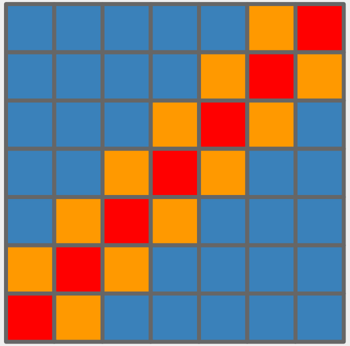

STORAGE
MAP
SHUFFLE
REDUCE
OUTPUT

Real Example

- A search engine company has crawled the internet, tracking **~1 trillion** *O(10^12)* pages.
- It tracks both when it shows these pages in search results and how many people have clicked on each page each week in the search results.
- If storing 1 MB *O(10^6)* of info for each page, storing 10^18 bytes = 1 exabyte!

Real Example

- The search engine wanted to understand the distribution of the mean/variance of the **click rate** for **domains**,e.g. *www.example.com/**
- Goal: **heatmap** with **mean** on x axis, **std. dev.** on y axis, z axis is count
- e.g. for Poisson-ish behaviour:



Complications

- Data storage infrastructure is **massively distributed** across disks, machines, and maybe even physical locations, as are the **computation** resources
- If it took 10^-3 seconds per page to perform the analysis, still looking at 10^9 seconds = many years of computing time!

Solution: MapReduce

MAP - for each URL

- Extract the domain, e.g. `http://nameremoved.com/2013/mip-callback.html` becomes `http://nameremoved.com`
- Check there is enough weeks of click data to be "meaningful", there has been at least one click, etc.
- Calculate click rate on this URL
- Emit key-value: <`domain, clickrate`> if everything OK, nothing otherwise

Solution: MapReduce

SHUFFLE

- Done by the MapReduce framework - no work required by user.
- Intermediate results don't fit in memory of any given machine! Key is ~ 30 bytes on average, value is ~8 bytes, so about 30 TB of intermediate data
- MapReduce framework will write intermediate results to disk in intelligent way
- Output: `<domain : [cr1,cr2,cr3,...]>`

Solution: MapReduce

REDUCE - for each domain

- Walk through values
  - Counting number of clickrate values = n
  - Accumulating sum of clickrate values = x
  - Accumulating sum of clickrate values squared = xsq
- Output key-value pair
  - `<domain : <n, mean, stddev>>`
  - mean `x/n`
  - stddev `sqrt(1/n * xsq - (x/n)^2)`

Still Not Done!

- We have now saved output of our MapReduce to disk, considerably smaller than our original data set but still large
- We now need another MapReduce that takes output of previous MapReduce as input.
- **Map**: “bins” the domains for our heatmap
- **Reduce**: counts domains in each bin

MapReduce: Nothing Too Fancy

- Like before:
  - We throw away a lot of information we don’t need
  - We iterate through the data line-by-line
  - We save temporary results to disk
  - Decompose operation into parts
  - The final data (heatmap counts) is usually **much** smaller than raw input data
- Difference from medium data: ***might not have data/compute all easily accessible.***

Aside: Map/Reduce

- "Map" and "Reduce" have alternative meaning in functional programming

```
function map(data, func)

    for i = 1 to length(data)
        data[i] = func(data[i])
    end
    return data
end
```

```
function reduce(data, func)
    accum = 0
    for i = 1 to length(data)
        accum = func(accum, data[i])
    end
    return accum
end
```

MapReduce Practice

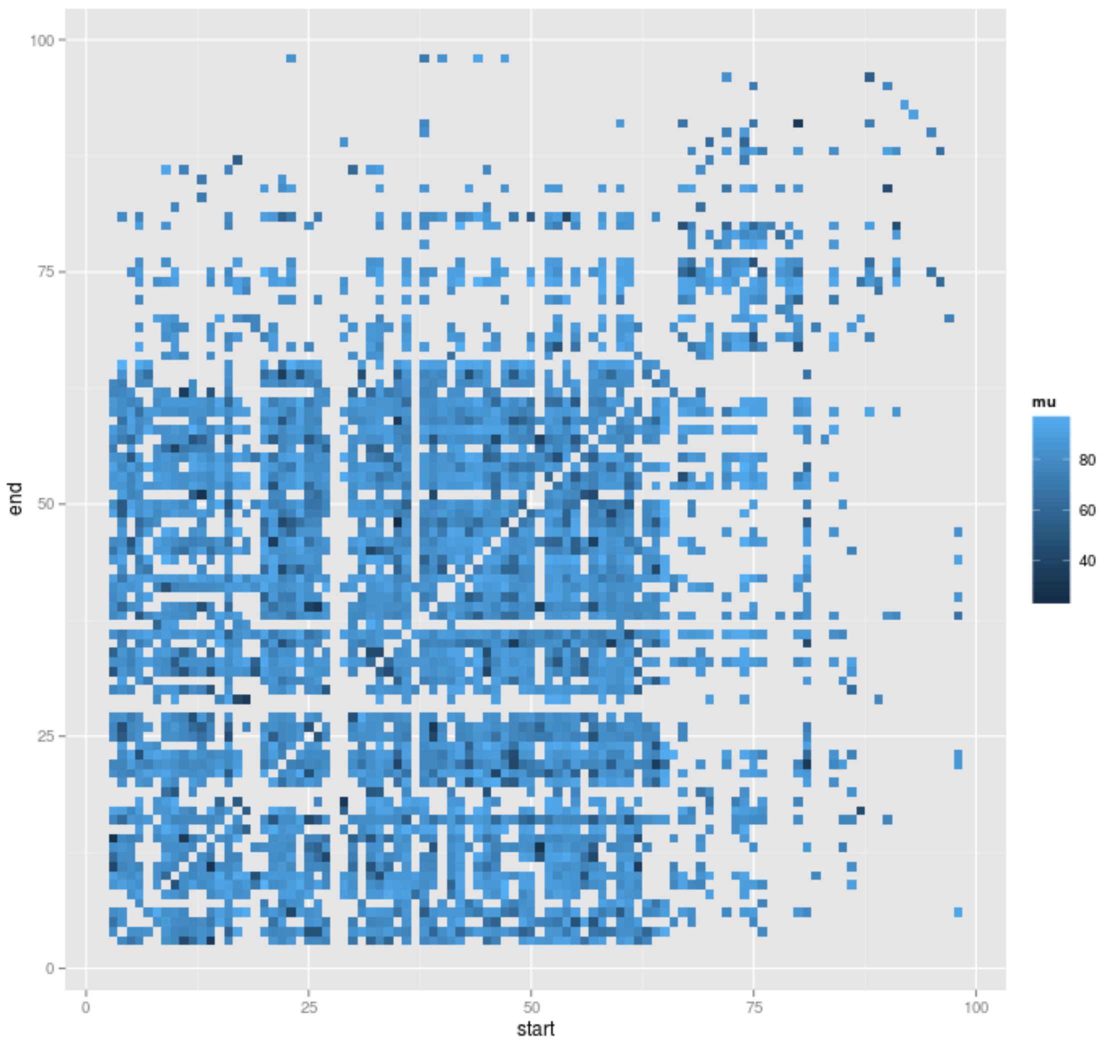
- We will practice thinking about a problem in a MapReduce way
- We will use the Julia `map` function and the Hubway data
- First, we will live code a simple example using trip data, then you will make one for the trips with a more advanced computation.

Live Coding

mapreduce_script.jl

Assignment 4 - Mappin’n’reducin’

- We will write code that calculates summary statistics about the flows of money instead of trips around the Hubway network
- See *assmt4_prompt.jl* for details
- Questions?



100
75
50
25
0
end
0
25
50
75
100
start
mu
80
60
40