Big Data and Security

Jeffrey Borowitz, PhD

Lecturer

Sam Nunn School of International Affairs

The Importance of Software: An Example of Search Engines

How Would You Make a Search Engine?

Let’s say you have all the web pages stored in a nice big table

How Would You Make a Search Engine?

- Let's say you have all the web pages stored in a nice big table
- You can start by looking through all the pages for your search term
  - Say you have 1 billion pages (which is the size of the web, to order of magnitude)
  - And you look through 1 page every microsecond
  - ⇒ It would take ~ 15 minutes to figure out which pages have your word
- You could split the pages and go faster with MapReduce
  - But it would still take 15 minutes of server time to do a search, which is expensive!
    - If Google did that, it would cost ~ $.05 per search, just for the hardware!
    - Interestingly, Google makes about $.025/search in revenue (2 trillion searches, $50 billion in revenue) so that can't be it!

The Answer: Indexing

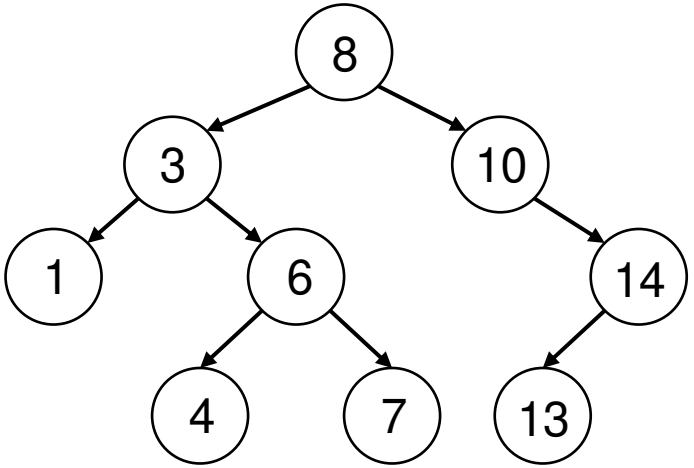
Go through the pages and make a table (called an “index”):

Words	Locations
bear	Page 1, Page 12, Page 14
cat	Page 3, Page 6

- Now how long will it take?
  - There are about 1 million words in English.
  - So you can look at only 1 million-ish words to see if they match “bear”
- This gets us down to ~ .5 seconds, which is way better!
- This is how indexes work in the back of books, too

But We Can Do Better!

- A tree!
- Sort all the search terms and put them in a tree like this
- Every tree node has the following property: all the nodes to the left are smaller, and all the nodes to the right are bigger
- It turns out, if you do this, you need only $\log_2 (1 \times 10^6) = 20$ operations!



Georgia Tech

Indexing Is A Key Technology

Indexing is used in a lot of places!

- When you log in to a website, it's used to look up your info
- When the airline looks up your reservation
- Etc. . .

Another Aspect of Search: Relevance Rankings

- When you search for things, you usually get rankings based on relevance.
- How is “relevance” measured?
  - One major method: term frequency - inverse document frequency.
  - To rank every document for its relevance to a query word,
    - Take the number of times that word appears in the document
    - Divide by the number of documents in your collection where that word appears
    - Do this for every document, and then sort by this score.
  - This ranking gives more weight to:
    - Terms that happen more often in a document (so “Congress” in articles about US politics)
    - Terms that appear less frequently in the documents you are searching (so the name of a specific Congressman in articles about US politics)

Relevance Rankings and Representing Data

Let's think about how we would calculate this term frequency on a computer

- If we're searching just the New York Times archives, there might be 50 articles/day * 365 days/year * 100 years ≈ 1.5 million documents.
- These documents might use something like 200,000 words

	Term Frequency		
Document	Aardvark	…	Zebra
1	0	…	2
…	…	…	…
1231249	1	…	0

Georgia Tech

Relevance Rankings and Representing Data

- This would take 1.5 million * 8 bytes * 200,000 ≈ 2 terabytes
- But our 1.5 million articles are probably on 10 kilobytes each, so all the data would only take 10 gigabytes.
- We're using 200x the whole data set to store this in the table!

	Term Frequency		
Document	Aardvark	…	Zebra
1	0	…	2
…	…	…	…
1231249	1	…	0

Representing Data

- The reason for this problem is the data is **sparse**
- Every row of our table has 200,000 columns, one for each word
- But each article is probably < 1,000 words long, so most of those values are 0.
- So we can do better with the "column store" style here:
  - This table is now 200,000 rows (for each word) and (if every document had all different words) 1,000 columns wide and ≈ 2 gigabytes
  - And if we get a search for a term, we can look it up and then read off all the document frequencies

Word	(Location, Frequency)
Aardvark	(doc 3, 4) (doc 1231249, 1)
…	…
Zebra	(doc 1, 2)

Georgia Tech

Relevance + Indexing = Modern Search

- It turns out that these relevance scoring plus index-based storage do a really good job of searching many documents
- A wonderful open source library called Lucene (visit this module's list of resources to access) basically implements this.
    - Lucene is the main codewhich implements what we've been talking about
- Google adds more pieces to their search (Naturally: Google spends $$$ on search technology, so this is a major simplification)
    - PageRank: a kind of popularity metric, where pages get importance based on how many other pages link to them (see this module's resource list)
    - Other measures of quality: recentcy, images
    - Input from human search evaluators

So What's the Point of This?

- Often, compared to a naïve way of solving a problem, things can be improved
  - We can store data in ways that take less space
  - We can store data in ways that make the particular questions we want to ask of the data easier to answer
- One of the major domains of academic computer scientists is to come up with stuff like:
  - Binary trees
  - Sparse data storage
- By some accounts, progress from things like data representation and clever algorithms is a bigger driver of improved computing than Moore's Law
  - A particular algorithm has sped up 43 million times, of which 1,000 times are due to hardware and 43,000 are due to software
  - Refer to the PDF in this module's resources (Box on page 71)

Lesson Summary

- How you structure data plays a huge role in system performance
- We talked through indexing as well as relevance scoring as they relate to modern search engines