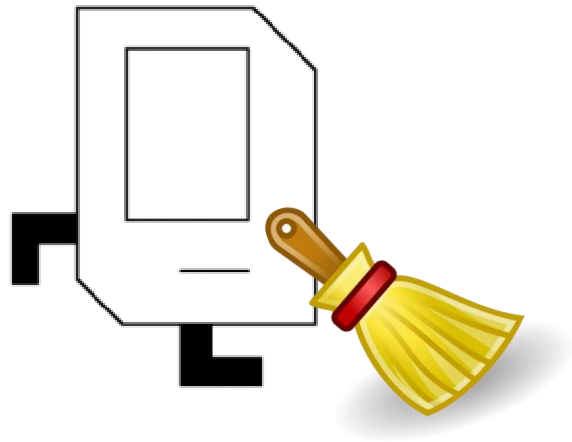
Search Engines

CS106A, Stanford University

Housekeeping



- Website for more practice Python problems:
  - https://codingbat.com/python
  - Built by our own Nick Parlante

Learning Goals

1. Learning about search engines
2. Getting some hints on Assignment #7

Search Engines

How to Build a Web Search Engine

- Crawling
  - Find relevant documents to search over
- Indexing
  - Record which terms appear in which documents
- Search
  - Determine which documents match user's query
- Ranking
  - Sort matching documents by "relevance" to user's query
- Serving
  - Infrastructure to get queries and give results
- Interface
  - User interface for presenting results to the user

In Assignment #7

- Crawling
  - We will provide document collection for you to search
- Indexing
  - You'll be writing this!
- Search
  - You'll be writing this!
- Ranking
  - Nothing fancy required, but great area for extensions
- Serving
  - Not required, but great area for extensions (more soon)
- Interface
  - Give you basic text interface, but great area for extensions

Indexing

- Inverted index (generally, just called an "index")
  - Similar to index in back of a book
  - For each word, you want to know where it is mentioned
- Mapping, where we have: term → list of documents containing that term
  - ***<u>Term</u>*** is the generic way we refer to a word, name, number, etc. that we might want to look up
- Consider the example:
  - Term "burrito" appears in the documents "recipes.txt", "greatest eats.txt", "top 10 foods.txt", and "favorites.txt"
  - Term "sushi" appears in documents "favorites.txt" and "Japanese foods.txt"
  - Term "samosa" appears in document "appetizers.txt"

Representing an Index in Python

- Consider the example:
  - term "burrito" appears in the documents "recipes.txt", "greatest eats.txt", "top 10 foods.txt", and "favorites.txt"
  - term "sushi" appears in documents "favorites.txt" and "Japanese foods.txt"
  - term "samosa" appears in document "appetizers.txt"
- In Python, use a dictionary to represent index
  - Map from term (key) to list of documents (value)

```
index = {
         'burrito': ['recipes.txt', 'greatest eats.txt',
                     'top 10 foods.txt', 'favorites.txt'],
         'sushi': ['favorites.txt', 'Japanese foods.txt'],
         'samosa': ['appetizers.txt']
        }
```

Building an Index in Assignment #7

- Given a set of documents
  - For each document, parse out all the terms:
    - Terms are separated from each other by space (or newline)
    - Terms should be converted to lowercase (for consistency)
    - Terms need to have punctuation stripped off start/end

```
>>> raw = '$$j.lo!'
>>> term = raw.strip(string.punctuation)
>>> term
'j.lo'
```

'doc1.txt':

```
*We* are 100,000
STRONG!  $$
```

- Example: Terms in 'doc1.txt':
  - `'*We*'` should be converted to term `'we'`
  - `'are'` should be converted to term `'are'`
  - `'100,000'` should be converted to term `'100,000'`
  - `'STRONG!'` should be converted to term `'strong'`
  - `'$$'` should be ignored. Punctuation by itself is not a term.

Building an Index in Assignment #7

'doc1.txt':

```
*We* are 100,000
STRONG!  $$
```

- Example: Terms in 'doc1.txt':
  - `'*We*'` should be converted to term `'we'`
  - `'are'` should be converted to term `'are'`
  - `'100,000'` should be converted to term `'100,000'`
  - `'STRONG!'` should be converted to term `'strong'`
  - `'$$'` should be ignored. Punctuation by itself is not a term.
- Resulting index (dictionary) in Python would be:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt']
}
```

Note: Python would print the dictionary all on one line. We just break it up on multiple lines in our examples for clarity.

Building an Index in Assignment #7

'doc2.txt':

```
Strong, you are!
--Yoda--
```

- Now, say we indexed 'doc2.txt':
  - `'Strong,'` should be converted to term `'strong'`
  - `'you'` should be converted to term `'you'`
  - `'are!'` should be converted to term `'are'`
  - `'--Yoda--'` should be converted to term `'yoda'`
- Updating our previous index with this data should give:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

A Final Note on Indexing

- Often, files have some information that we want to keep track of (such as a title) for later display
  - Here, first line of each file contains a title that we want to keep track of
  - The terms in the title line should still be indexed like every other line in the file
- Build a mapping (dictionary) from file names to titles (for later display):

```
{
 'quote1.txt': 'Yoda quote',
 'quote2.txt': "Gandhi's wisdom"
}
```

'quote1.txt':

```
Yoda quote

Strong, you are!
--Yoda--
```

'quote2.txt':

```
Gandhi's wisdom

Be the change
that you wish to
see in the
world.
--Mahatma Gandhi
```

Note: in the index of these files, **"gandhi's"** would be a term (with the apostrophe embedded) since the apostrophe is not at the end beginning/end of the term.

Search

- Once you have an index, searching is straightforward
  - In the user interface, user enters a query
    - Note: Terms in query will be separated by spaces and converted to lowercase. (Can assume no punctuation before/after query terms.)
  - For each term in query, we use the index to look up the list of documents that the term appears in
    - This list of documents is called a "posting list"
- For one term queries, the posting list from the index directly provides the results to the query
- For multi-term queries, the way you combine posting lists for each term determines how the search works

Multi-Term Queries

- Can <u>add together</u> the results (uniquely) of all the posting lists
  - This would be comparable to doing a **<u>union</u>** with sets
  - This corresponds to treating the query as a *disjunction*
    - We return any document that contains **<u>any</u>** of the terms in query
    - Logically, it's like using the connective "OR" between query terms
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "yoda strong"

Posting list:

Multi-Term Queries

- Can <u>add together</u> the results (uniquely) of all the posting lists
  - This would be comparable to doing a **<u>union</u>** with sets
  - This corresponds to treating the query as a *disjunction*
    - We return any document that contains **<u>any</u>** of the terms in query
    - Logically, it's like using the connective "OR" between query terms
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "yoda strong"

Posting list:

```
['doc2.txt']
```

Multi-Term Queries

- Can <u>add together</u> the results (uniquely) of all the posting lists
  - This would be comparable to doing a **<u>union</u>** with sets
  - This corresponds to treating the query as a *disjunction*
    - We return any document that contains **<u>any</u>** of the terms in query
    - Logically, it's like using the connective "OR" between query terms
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "yoda strong"

Posting list:

```
['doc2.txt', 'doc1.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "are you yoda"

Posting list:

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "are you yoda"

Posting list:

```
['doc1.txt', 'doc2.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an <u>**intersection**</u> with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain <u>**every**</u> term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "are you yoda"

Posting list:

```
['doc2.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "are you yoda"

Posting list:

```
['doc2.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "we are yoda"

Posting list:

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "we are yoda"

Posting list:

```
['doc1.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an **<u>intersection</u>** with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain **<u>every</u>** term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "we are yoda"

Posting list:

```
['doc1.txt']
```

Multi-Term Queries

- Can <u>take the overlap</u> of the results (uniquely) of all the posting lists
  - This would be comparable to doing an <u>**intersection**</u> with sets
  - This corresponds to treating the query as a *conjunction*
    - We return documents that contain <u>**every**</u> term in query
    - Logically, it's like using the connective "AND" between query terms
  - This is what you'll implement for Assignment #7
  - Recall index:

```
{
 'we': ['doc1.txt'],
 'are': ['doc1.txt', 'doc2.txt'],
 '100,000': ['doc1.txt'],
 'strong': ['doc1.txt', 'doc2.txt'],
 'you': ['doc2.txt'],
 'yoda': ['doc2.txt']
}
```

  - Query: "we are yoda"

Posting list:

```
[]
```

Let's take it out for a spin:
searchengine.py

Ranking Documents

- In Assignment #7, you just display the documents that are considered matches to the query
  - You are not ranking them in any particular order
  - But, this is an area for cool extensions, so let's chat about it…
- One of the richest research areas in search is how to rank documents (i.e., sort them by relevance to user)
  - Doing this requires that we keep track of more information in the index (e.g., store lists/tuples rather than just file names)
  - Examples of additional information that's useful for ranking:
    - Number of times a term appears in a document
    - The positions of the terms in each document
    - How rare particular terms are in the whole collection of documents
    - How "popular" a document is (e.g., analyze link structure on the web)

Measures of Textual Similarity

- Classic approach: Documents/query similarity is a function of *term frequency within the document* and *across all documents*
- TF(w) = frequency of term w in a document/query
  - Intuition: a word appearing more frequently in a document is more likely to be related to its "meaning"
- IDF(w) = log $(N/n_w) + 1$

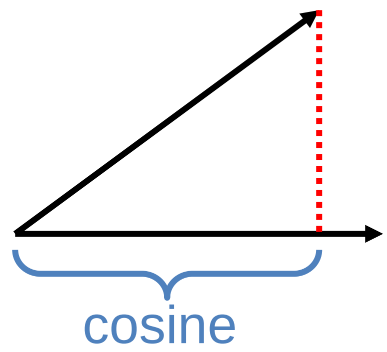
  where N = total # documents, n_w is # documents containing w
  - Intuition: words that appear in many documents (e.g., "the") are generally not very informative/contentful terms
- TFIDF: contribution of each term is product of these:

  TFIDF(w) = TF(w) x IDF(w)

Using TFIDF to Measure Similarity

- Consider each document as a list/vector:

	dog	compute	window	...
Doc. 1 = [	3.2,	0,	1.2,	...]
Doc. 2 = [	0,	2.1,	5.4,	...]
Doc. 3 = [	0,	1.7,	0,	...]



- Lists/vectors are constructed such that
  - Each element of list/vector represents a term w_i
  - Each element of list/vector has value: TFIDF(w_i)
  - Normalize the vectors to unit length (using Euclidean norm)
- Document similarity to another document or query is measured using the cosine between the TFIDF vectors of the documents/queries
  - Cosine = vector dot product
  - Called "Vector Space Model"

Learning Goals

1. Learning about search engines
2. Getting some hints on Assignment #7

Bonus story time:

Google

(...before it was Google)

STANFORD COMPUTER FORUM

TWENTY-NINTH ANNUAL MEETING

MARCH 19-20, 1997

Department of Computer Science
Professor Jean-Claude Latombe, Chair

Thursday, March 20, 1997

1:30-3:00	**Parallel Session III-A: Information Retrieval** Professor Rajeev Motwani, Chair H-P Auditorium
1:30	**Information Retrieval and the Web** Larry Page — Professor Terry Winograd, Advisor
2:00	**Creating Personalized Yahoo!'s: Automated Hierarchical Clustering and Classification of Documents** Mehran Sahami — Professor Daphne Koller, Advisor
2:30	**SenseMaker: An Information-Exploration Interface** Michelle Baldonado — Professor Terry Winograd, Advisor
3:00-3:15	**Break**

Thursday, March 20, 1997

10:30-12:00 **Parallel Session II-A: Data Mining**
Professor Nils Nilsson, Chair
NEC Auditorium

10:40 **Adaptive Web Page Recommendation**
Marko Balabanovic — Professor Yoav Shoham, Advisor

11:05 **Problems in Data Mining**
Sergey Brin — Professor Hector Garcia-Molina, Advisor

11:30 **Association Rules**
Craig Silverstein — Professor Rajeev Motwani, Advisor

12:00-1:30 **Lunch**
Gates Building, Room 104

Wednesday, March 19, 1997

8:30-9:00 **Registration and Continental Breakfast**
Gates Building, Basement Lobby

9:00-10:30 **Opening Session**
Gates Building, H-P Auditorium

Welcoming Remarks
Carolyn Tajnai, Director, Computer Forum
Professor Yoav Shoham, Annual Meeting Program Chair

Department Greetings
Professor Jean-Claude Latombe, Chairman, Computer Science Department
William F. Miller, Computer Forum Faculty Chair

9:30 **Keynote Address**
Dr. Eric Schmidt, CTO, CEO, Sun Microsystems
Evolution or Revolution? The Future of Network Computing

10:30-11:00 **Break**

Google's Beginnings

- In mid-1990's, Larry Page and Sergey Brin did research as part of the Stanford Digital Library project
  - Original project was called "BackRub"
- Large parts of Google were originally built in Python
  - Here's some of that code (it's written in Python 1.4)

```
class RobotFileParser:

  def __init__(self):
      self.rules = {}

  def parse(self, lines):
      active = []
      for line in lines:
          # blank line terminates current record
          if not line[:-1]:
              active = []
              continue
          # remove optional comment and strip line
          line = string.strip(line[:string.find(line, '#')])
    …
```

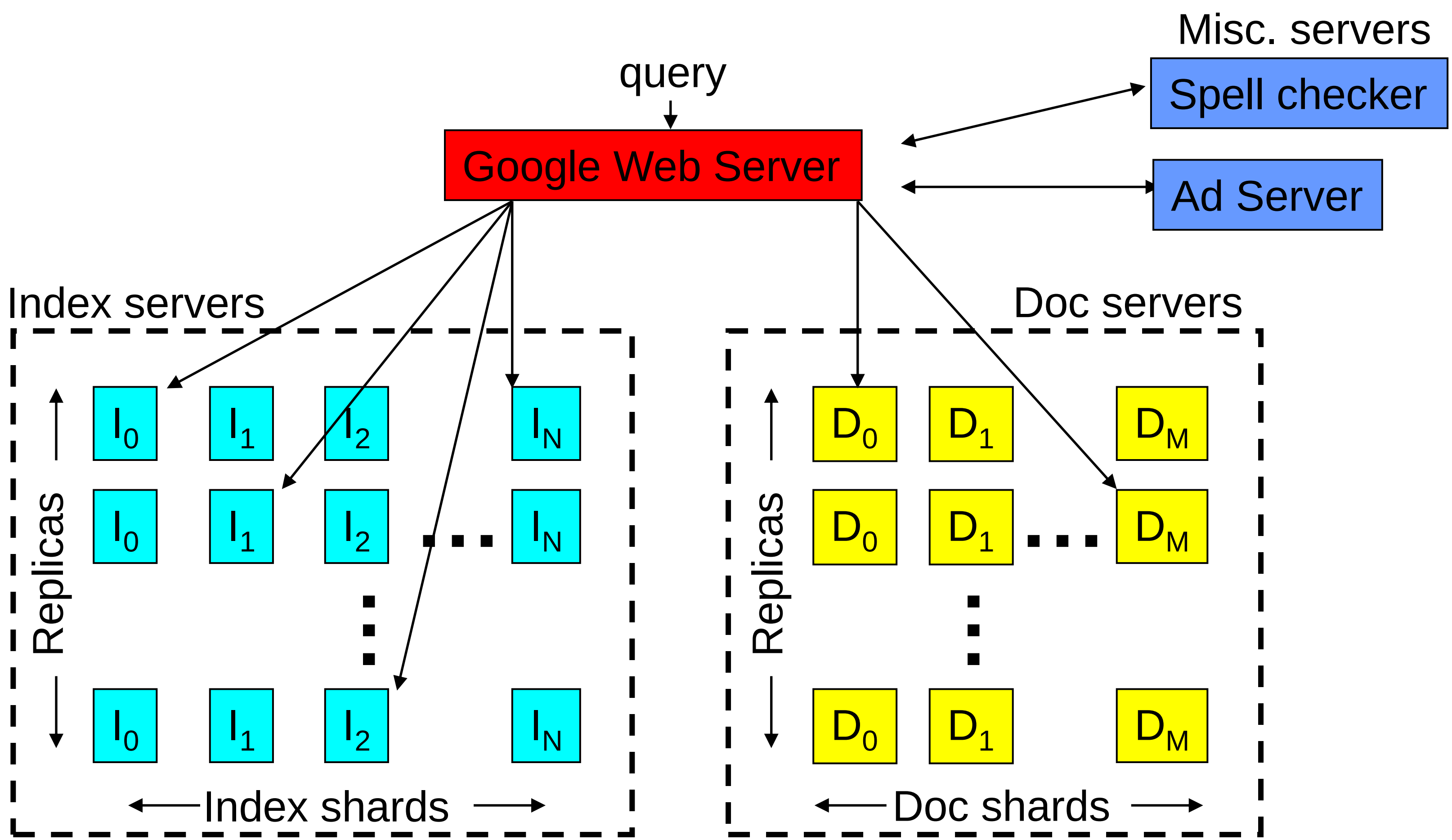
http://google.stanford.edu

Image courtesy of Google

Google's Index (circa 2004)

- Too large to fit in memory for one machine
- Split index into pieces, called *shards*
  - Shards are small enough to have several per machine
  - Replicate the shards for robustness
- Need to still store original documents
  - Want to show users "snippets" of query terms in context
  - Use same sharding concept to store original documents
- Replicate this whole structure within/across data centers

Google Infrastructure (circa 2004)



Elapsed time: 0.25s, machines involved: 1000+

google.stanford.edu (circa 1997)

Image courtesy of Google

google.com (1999)

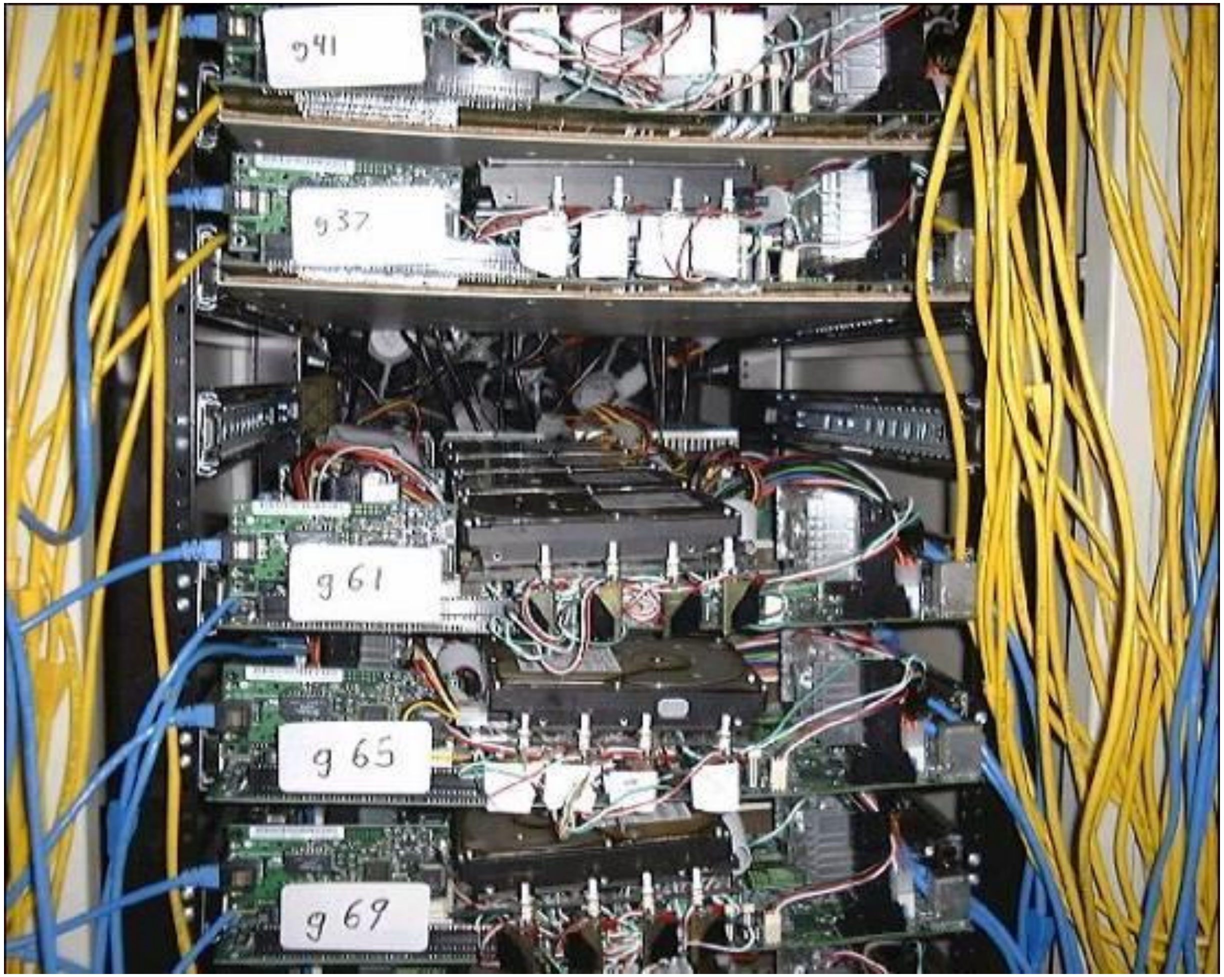


Image courtesy of Google

Google Data Center (circa 2000)

Image courtesy of Google

Empty Google Data Center (2001)

Image courtesy of Google

3 Days Later…

Image courtesy of Google

A Day in the Life of Google

A picture is worth a few hundred million search queries…

Image courtesy of Google