Distributed Systems

17. Case study: [Original] Google Cluster Architecture

Paul Krzyzanowski

Rutgers University

Fall 2018

A note about relevancy

This describes the Google search cluster architecture in the mid 2000s. The search infrastructure was overhauled in 2010.

Nevertheless, the lessons are still valid and this demonstrates how incredible scalability has been achieved using commodity computers by exploiting parallelism.

What is needed?

- A single Google search query
  - Reads hundreds of megabytes of data
  - Uses tens of billions of CPU cycles
- Environment needs to support tens of thousands of queries per second
- Environment must be
  - Fault tolerant
  - Economical (price-performance ratio matters)
  - Energy efficient (this affects price; watts per unit of performance matters)
- Workload should be highly parallelizable
  - CPU performance matters less than price/performance ratio

Key design principles

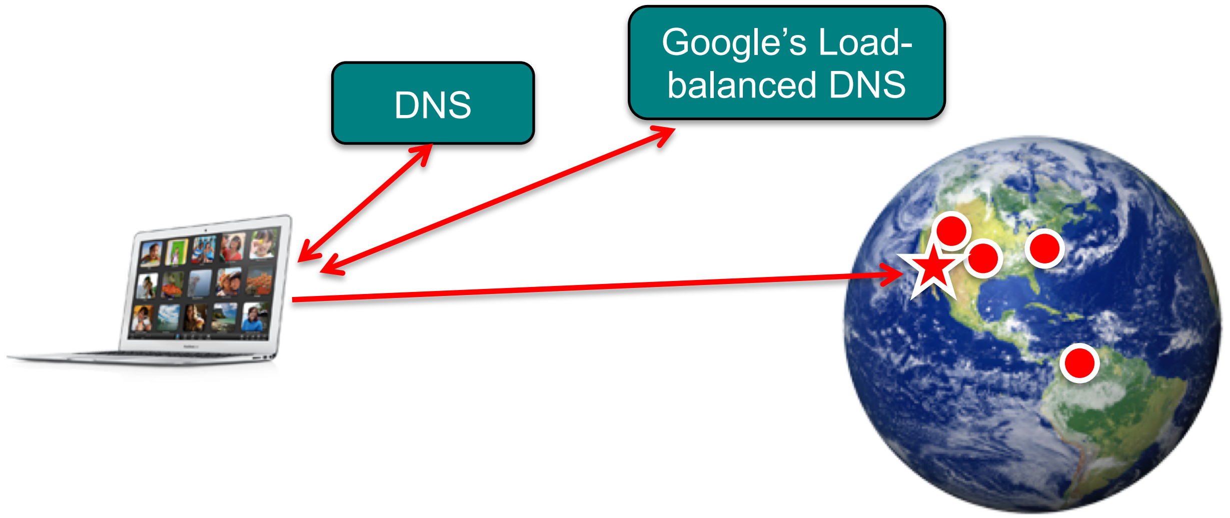
- Have <u>reliability reside in software</u>, not hardware
  - Use low-cost (unreliable) commodity PCs to build a high-end cluster
  - Replicate services across machines & detect failures

- Design for <u>best total throughput</u>, not peak server response time
  - Response time can be controlled by parallelizing requests
  - Rely on replication: this helps with availability too

- <u>Price/performance ratio</u> more important than peak performance

Step 1: DNS

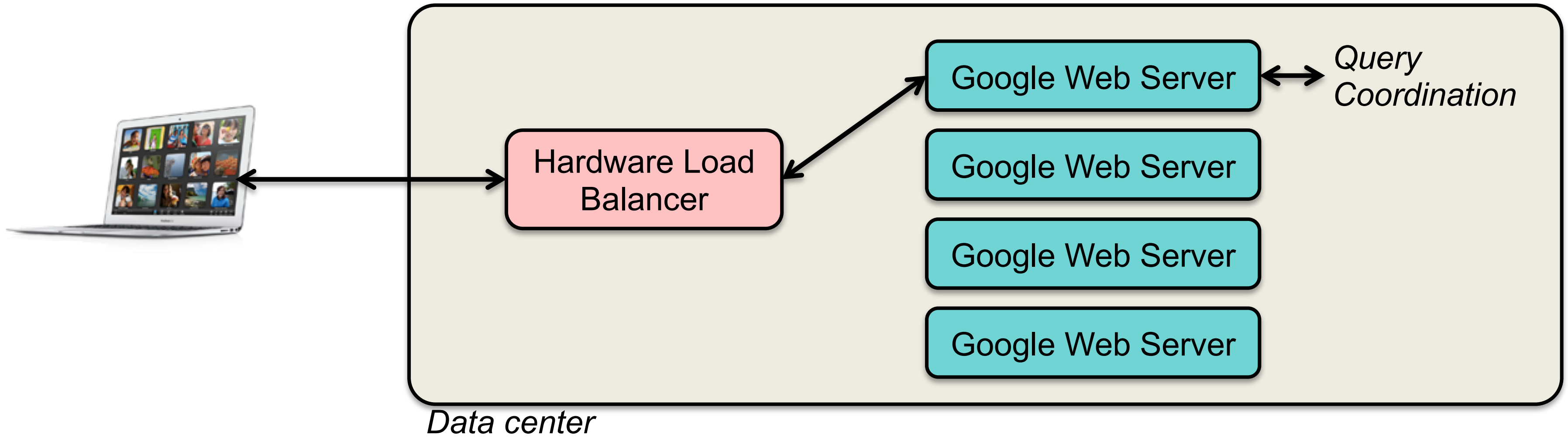
- User's browser must map google.com to an IP address
- "google.com" comprises multiple clusters distributed worldwide
  - Each cluster contains thousands of machines
- DNS-based load balancing
  - Select cluster by taking user's geographic & network proximity into account
  - Load balance across clusters



1. Contact DNS server(s) to find the DNS server responsible for google.com
2. Google's DNS server returns addresses based on location of request
3. Contact the appropriate cluster

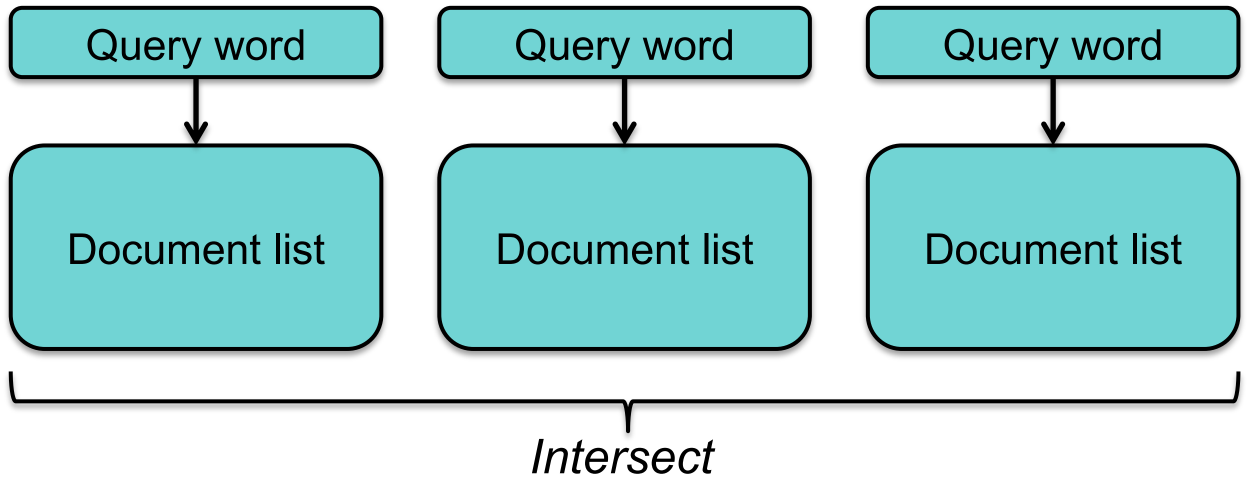
Step 2: Send HTTP request

- IP address corresponds to a load balancer within a cluster
- Load balancer
  - Monitors the set of Google Web Servers (GWS)
  - Performs local load balancing of requests among available servers
- GWS machine receives the query
  - Coordinates the execution of the query
  - Formats results into an HTML response to the user



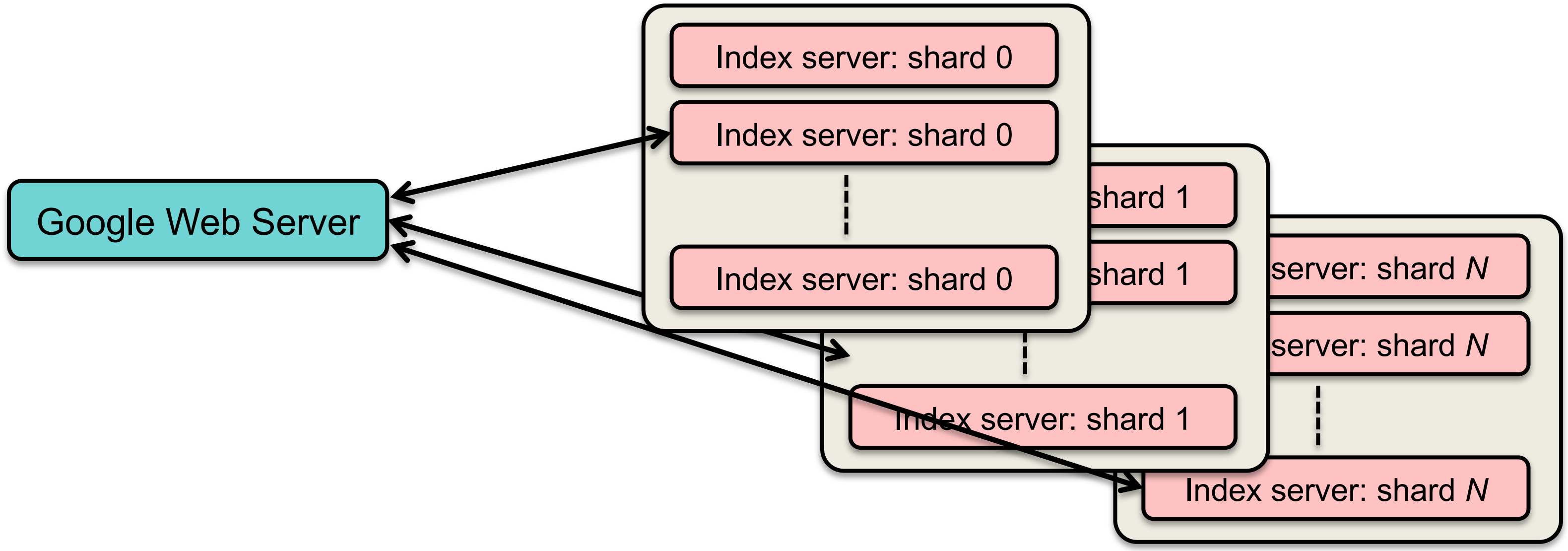
Step 3. Find documents via inverted index

- Index Servers
  - Map each query word → {list of documents} *(this is the* ***hit list****)*
    - Inverted index generated from web crawlers → MapReduce
  - Intersect the hit lists of each per-word query
    - Compute relevance score for each document
    - Determine set of documents
    - Sort by relevance score

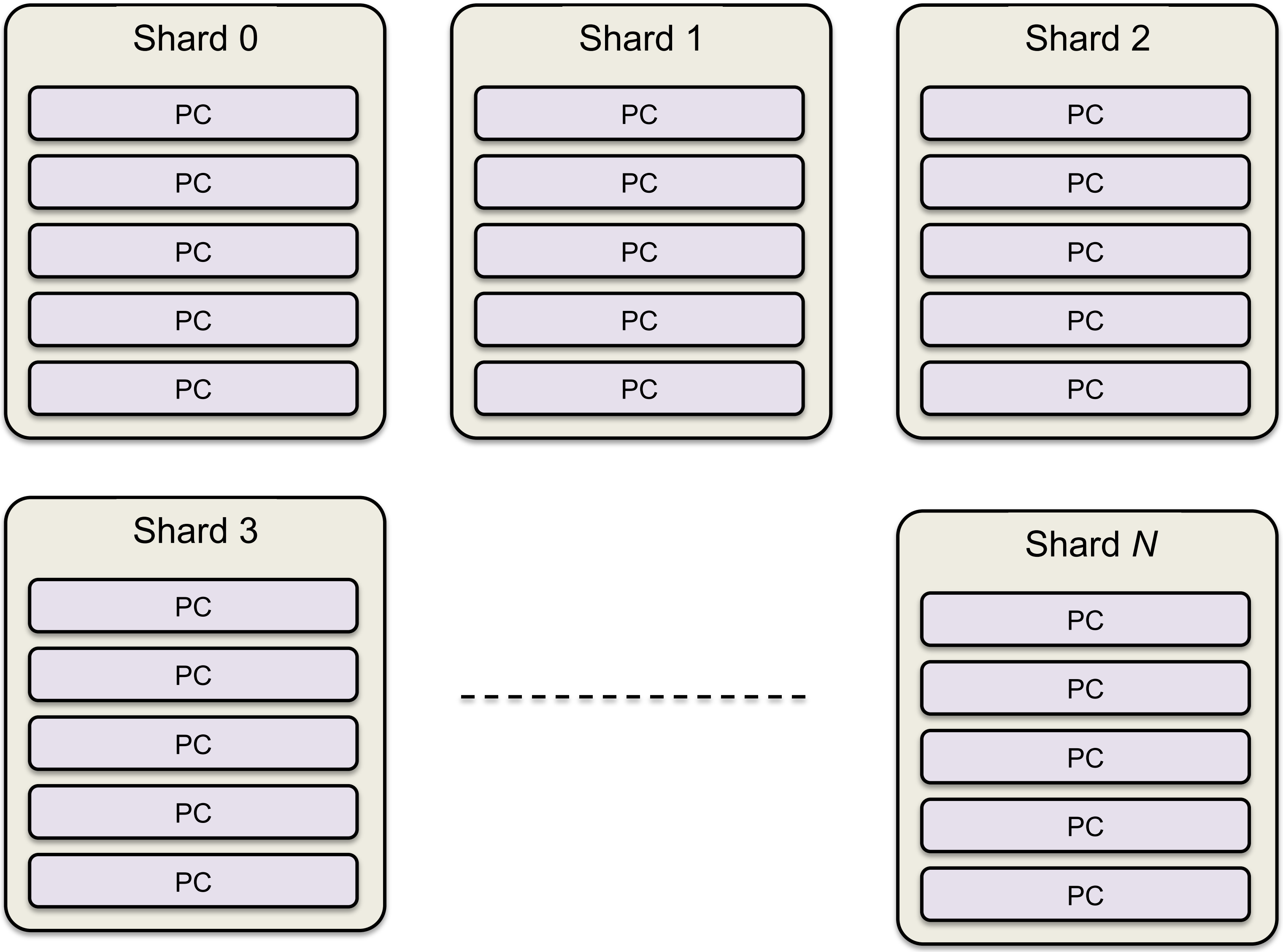


Parallel search through an inverted index

- Inverted index is 10s of terabytes
- Search is parallelized
  - Index is divided into *index shards*
    - Each index shard is built from a randomly chosen subset of documents
    - Pool of machines serves requests for each shard
    - Pools are load balanced
  - Query goes to one machine per pool responsible for a shard
- Final result is ordered list of document identifiers (*docids*)



Index Servers



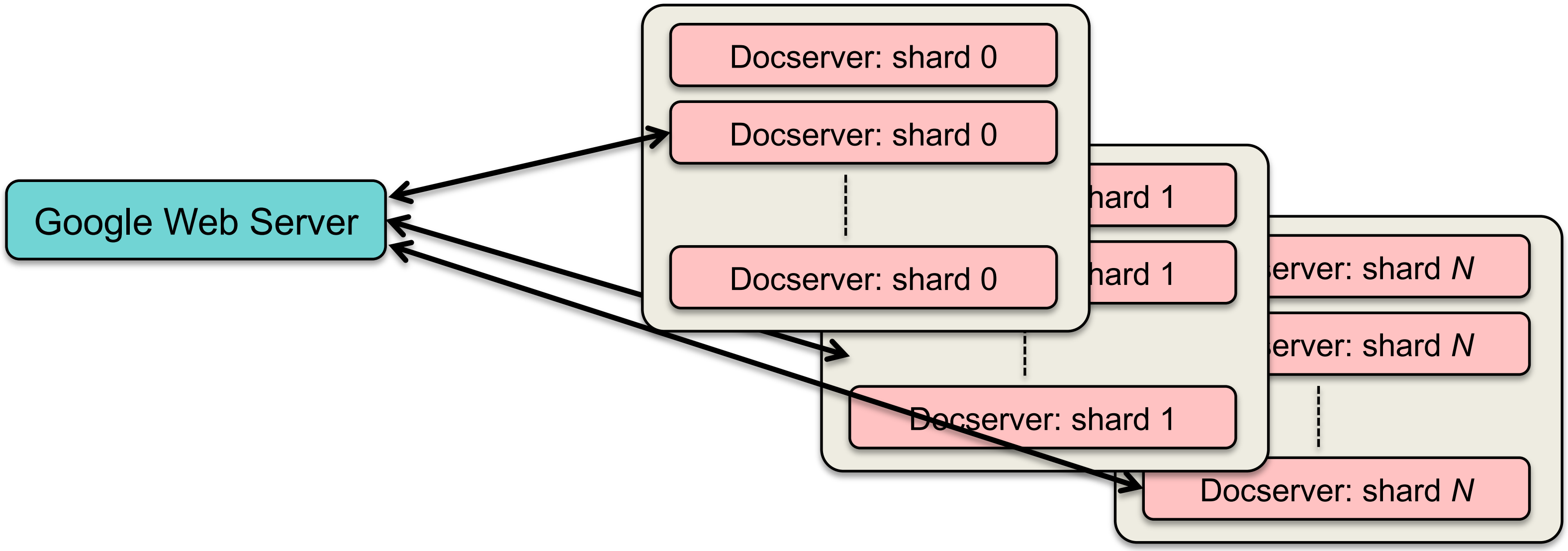
© 2013-2018 Paul Krzyzanowski

Step 4. Get title & URL for each docid

- For each docid, the GWS looks up
  - Page title
  - URL
  - Relevant text: document summary specific to the query
- Handled by document servers (docservers)

Parallelizing document lookup

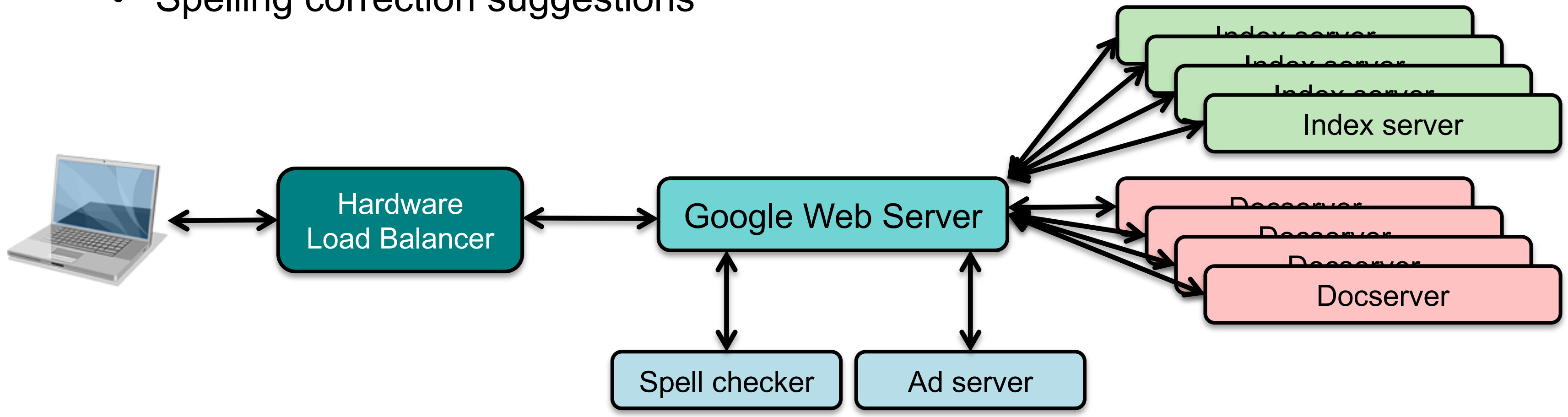
- Like index lookup, document lookup is partitioned & parallelized
- Documents distributed into smaller shards
  - Each shard = subset of documents
- Pool of load-balanced servers responsible for processing each shard
- Together, document servers access a cached copy of the entire web!



© 2013-2018 Paul Krzyzanowski

Additional operations

- In parallel with search:
  - Send query to a spell-checking system
  - Send query to an ad-serving system to generate ads
- When all the results are in, GWS generates HTML output:
  - Sorted query results
    - With page titles, summaries, and URLs
    - Ads
    - Spelling correction suggestions



Lesson: exploit parallelism

- Instead of looking up matching documents in a large index
  - Do many lookups for documents in smaller indices
  - Merge results together: merging is simple & inexpensive
- Divide the stream of incoming queries
  - Among geographically-distributed clusters
  - Load balance among query servers within a cluster
- Linear performance improvement with more machines
  - Shards don't need to communicate with each other
  - Increase # of shards across more machines to improve performance

The End