Challenges in Building Large-Scale Information Retrieval Systems

Jeff Dean
Google Fellow
jeff@google.com

Why Work on Retrieval Systems?

- Challenging blend of science and engineering
  - Many interesting, unsolved problems
  - Spans many areas of CS:
    - architecture, distributed systems, algorithms, compression, information retrieval, machine learning, UI, etc.
  - Scale far larger than most other systems

- Small teams can create systems used by hundreds of millions

Retrieval System Dimensions

- Must balance engineering tradeoffs between:
  - number of documents indexed
  - queries / sec
  - index freshness/update rate
  - query latency
  - information kept about each document
  - complexity/cost of scoring/retrieval algorithms
- Engineering difficulty roughly equal to the product of these parameters
- All of these affect overall performance, and performance per $

1999 vs. 2009

- # docs: ~70M to many billion
- queries processed/day:
- per doc info in index:
- update latency: months to minutes
- avg. query latency: <1s to <0.2s

- More machines * faster machines:

1999 vs. 2009

- # docs: ~70M to many billion ~100X
- queries processed/day:
- per doc info in index:
- update latency: months to minutes
- avg. query latency: <1s to <0.2s

- More machines * faster machines:

1999 vs. 2009

- # docs: ~70M to many billion **~100X**
- queries processed/day: **~1000X**
- per doc info in index:
- update latency: months to minutes
- avg. query latency: <1s to <0.2s

- More machines * faster machines:

Google

1999 vs. 2009

- # docs: ~70M to many billion **~100X**
- queries processed/day: **~1000X**
- per doc info in index: **~3X**
- update latency: months to minutes
- avg. query latency: <1s to <0.2s

- More machines * faster machines:

Google

1999 vs. 2009

- # docs: ~70M to many billion **~100X**
- queries processed/day: **~1000X**
- per doc info in index: **~3X**
- update latency: months to minutes **~10000X**
- avg. query latency: <1s to <0.2s

- More machines * faster machines:

1999 vs. 2009

- # docs: ~70M to many billion **~100X**
- queries processed/day: **~1000X**
- per doc info in index: **~3X**
- update latency: months to minutes **~10000X**
- avg. query latency: <1s to <0.2s **~5X**

- More machines * faster machines:

1999 vs. 2009

- # docs: ~70M to many billion **~100X**
- queries processed/day: **~1000X**
- per doc info in index: **~3X**
- update latency: months to minutes **~10000X**
- avg. query latency: <1s to <0.2s **~5X**

- More machines * faster machines: **~1000X**

Google

Constant Change

- Parameters change over time
  - often by many orders of magnitude
- Right design at X may be very wrong at 10X or 100X
  - ... design for ~10X growth, but plan to rewrite before ~100X
- Continuous evolution:
  - 7 significant revisions in last 10 years
  - often rolled out without users realizing we've made major changes

Rest of Talk

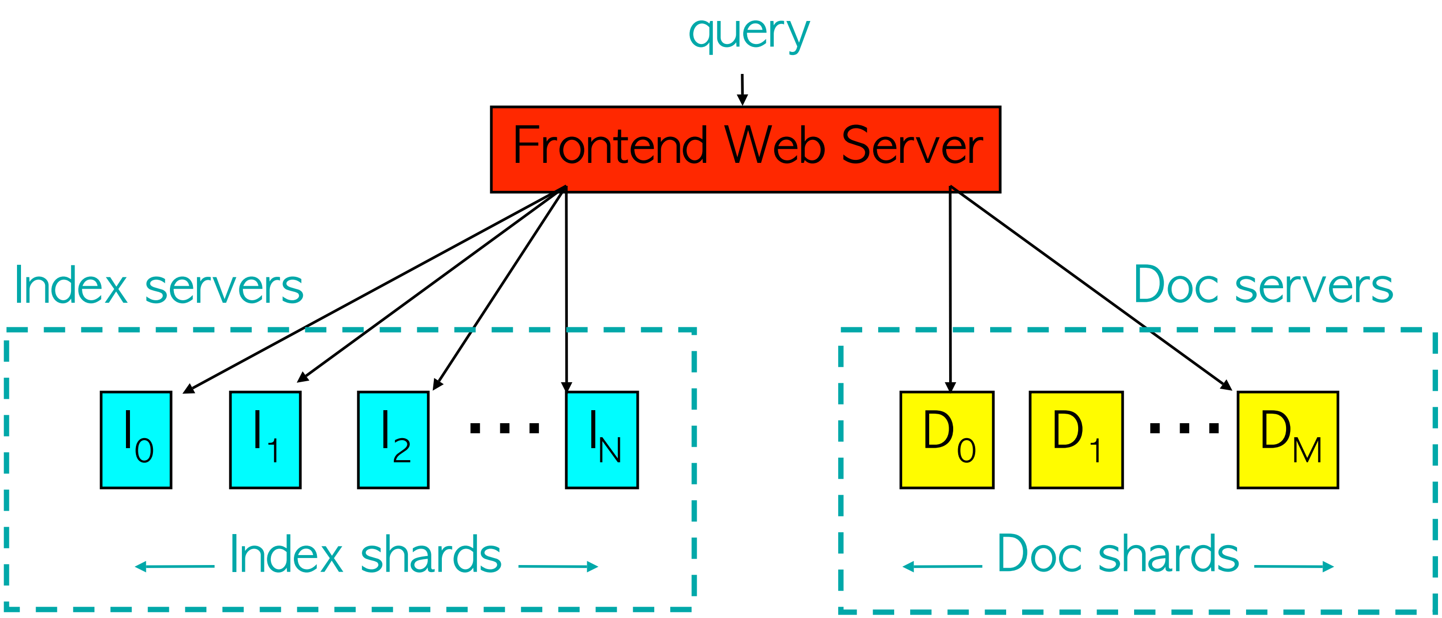
- Evolution of Google's search systems
  - several gens of crawling/indexing/serving systems
  - brief description of supporting infrastructure

  - Joint work with many, many people

- Interesting directions and challenges

"Google" Circa 1997 (google.stanford.edu)

Research Project, circa 1997



Google

Ways of Index Partitioning

- **By doc**: each shard has index for subset of docs
  - pro: each shard can process queries independently
  - pro: easy to keep additional per-doc information
  - pro: network traffic (requests/responses) small
  - con: query has to be processed by each shard
  - con: O(K*N) disk seeks for K word query on N shards

Ways of Index Partitioning

- **By doc**: each shard has index for subset of docs
  - pro: each shard can process queries independently
  - pro: easy to keep additional per-doc information
  - pro: network traffic (requests/responses) small
  - con: query has to be processed by each shard
  - con: O(K*N) disk seeks for K word query on N shards
- **By word**: shard has subset of words for all docs
  - pro: K word query => handled by at most K shards
  - pro: O(K) disk seeks for K word query
  - con: much higher network bandwidth needed
    - data about each word for each matching doc must be collected in one place
  - con: harder to have per-doc information

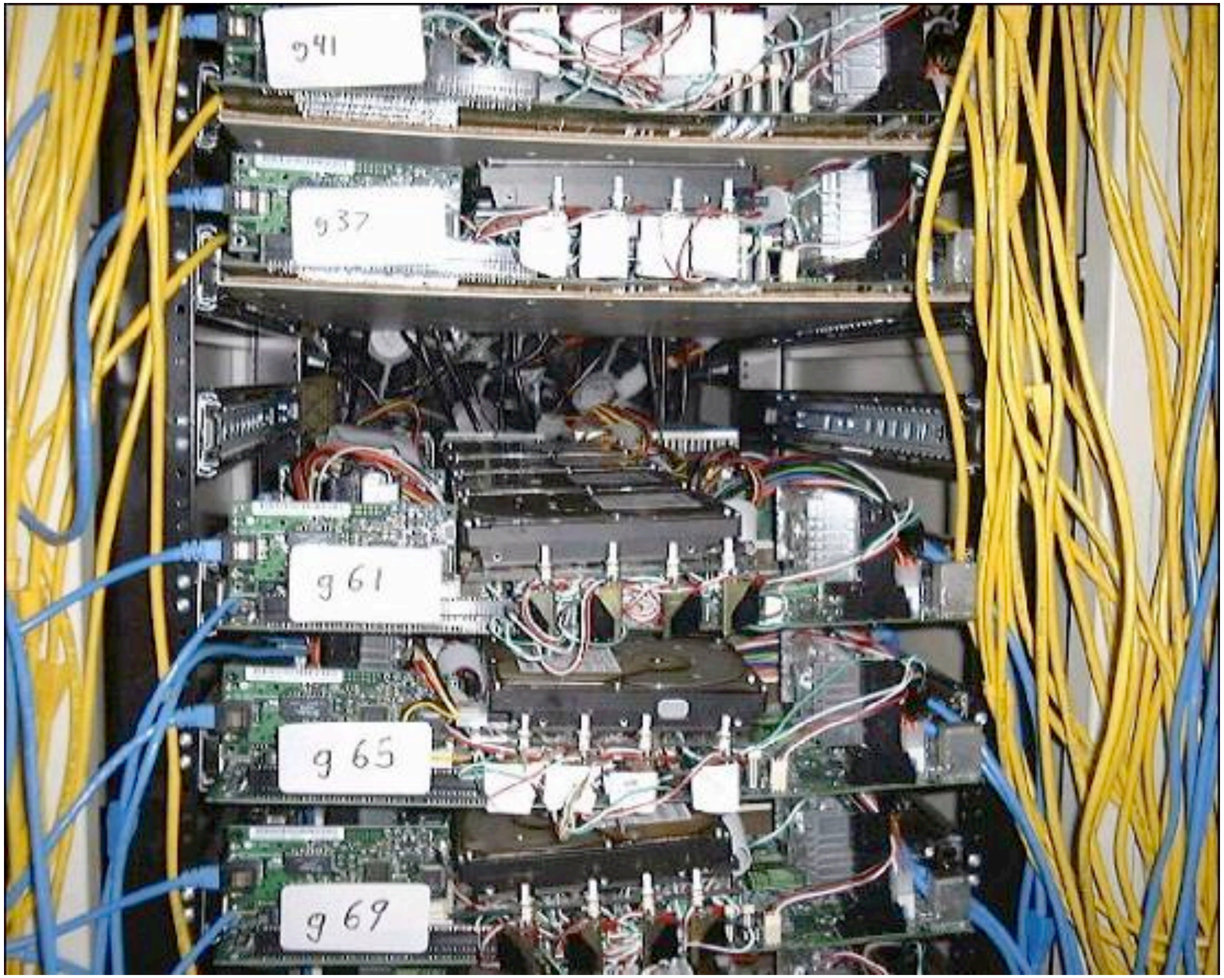
Ways of Index Partitioning

In our computing environment, by doc makes more sense

Basic Principles

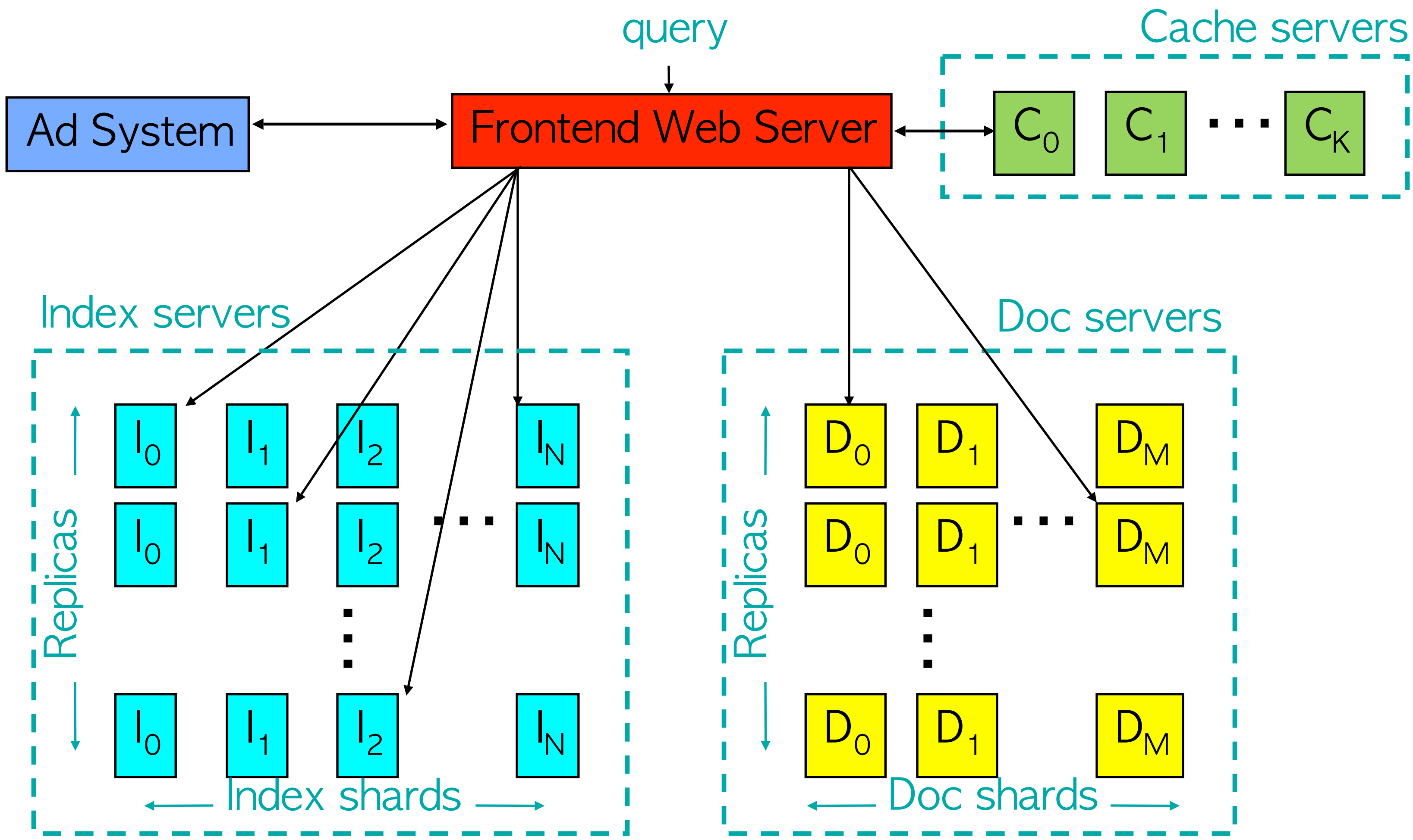
- Documents assigned small integer ids (docids)
  - good if smaller for higher quality/more important docs
- Index Servers:
  - given (query) return sorted list of (score, docid, ...)
  - partitioned ("sharded") by docid
  - index shards are replicated for capacity
  - cost is O(# queries * # docs in index)
- Doc Servers
  - given (docid, query) generate (title, snippet)
  - map from docid to full text of docs on disk
  - also partitioned by docid
  - cost is O(# queries)

“Corkboards” (1999)



Google

Serving System, circa 1999



Caching

- Cache servers:
  - cache both index results and doc snippets
  - hit rates typically 30-60%
    - depends on frequency of index updates, mix of query traffic, level of personalization, etc
- Main benefits:
  - performance! 10s of machines do work of 100s or 1000s
  - reduce query latency on hits
    - queries that hit in cache tend to be both popular and expensive (common words, lots of documents to score, etc.)
- Beware: big latency spike/capacity drop when index updated or cache flushed

Crawling (circa 1998-1999)

- Simple batch crawling system
  - start with a few URLs
  - crawl pages
  - extract links, add to queue
  - stop when you have enough pages
- Concerns:
  - don't hit any site too hard
  - prioritizing among uncrawled pages
    - one way: continuously compute PageRank on changing graph
  - maintaining uncrawled URL queue efficiently
    - one way: keep in a partitioned set of servers
  - dealing with machine failures

Indexing (circa 1998-1999)

- Simple batch indexing system
  - Based on simple unix tools
  - No real checkpointing, so machine failures painful
  - No checksumming of raw data, so hardware bit errors caused problems
    - Exacerbated by early machines having no ECC, no parity
    - Sort 1 TB of data without parity: ends up "mostly sorted"
    - Sort it again: "mostly sorted" another way

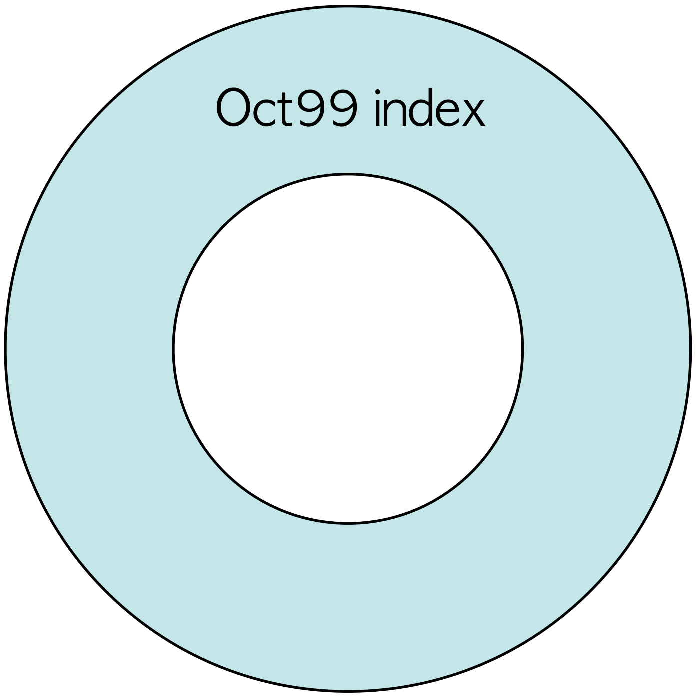
- “Programming with adversarial memory”
  - Led us to develop a file abstraction that stored checksums of small records and could skip and resynchronize after corrupted records

Index Updates (circa 1998-1999)

- 1998-1999: Index updates (~once per month):
  - Wait until traffic is low
  - Take some replicas offline
  - Copy new index to these replicas
  - Start new frontends pointing at updated index and serve some traffic from there

Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

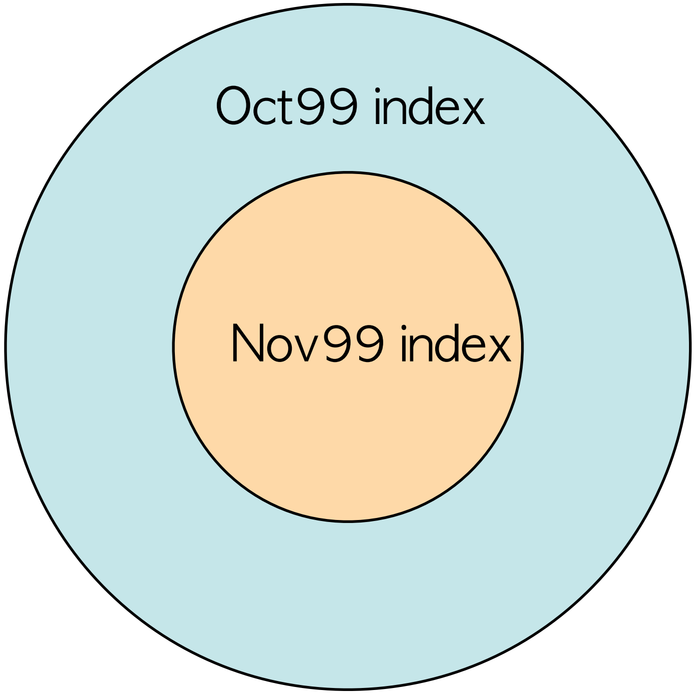


Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

1. Copy new index to inner half of disk (while still serving old index)

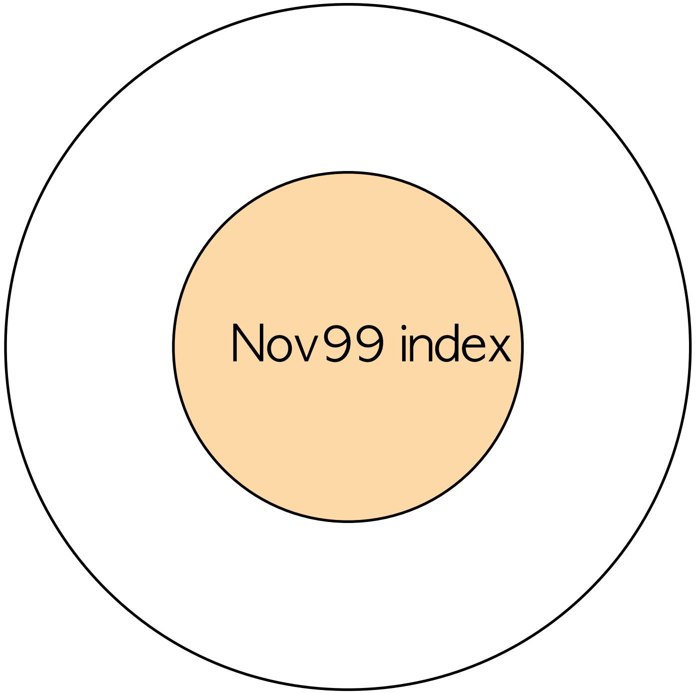
2. Restart to use new index



Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

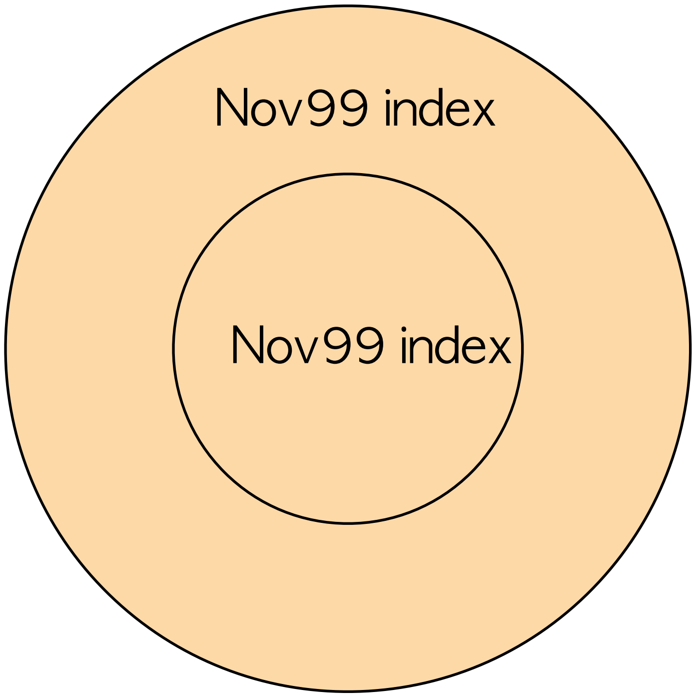
1. Copy new index to inner half of disk (while still serving old index)
2. Restart to use new index
3. Wipe old index



Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

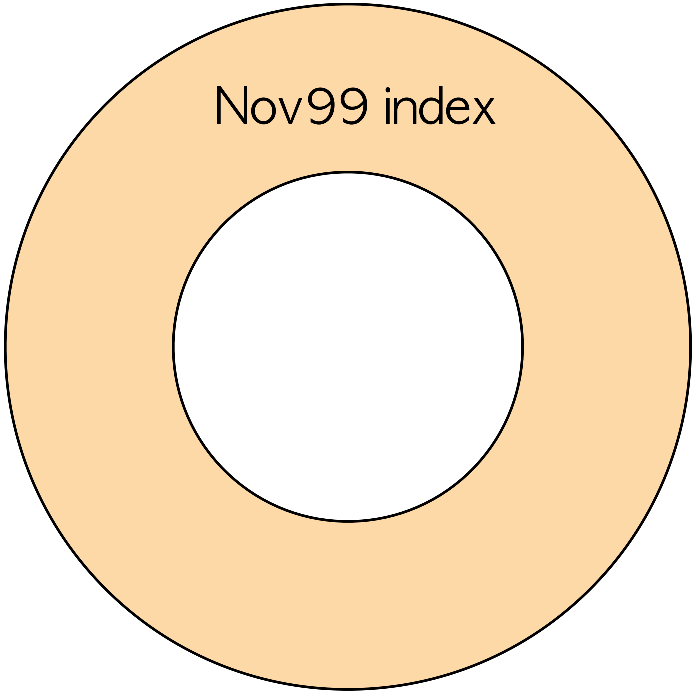
1. Copy new index to inner half of disk (while still serving old index)
2. Restart to use new index
3. Wipe old index
4. Re-copy new index to faster half of disk



Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

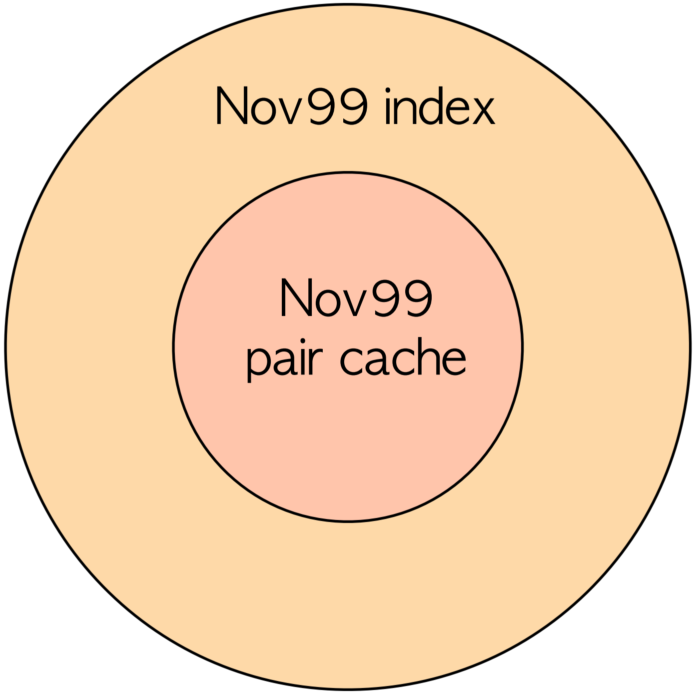
1. Copy new index to inner half of disk (while still serving old index)
2. Restart to use new index
3. Wipe old index
4. Re-copy new index to faster half of disk
5. Wipe first copy of new index



Index Updates (circa 1998-1999)

- Index server disk:
  - outer part of disk gives higher disk bandwidth

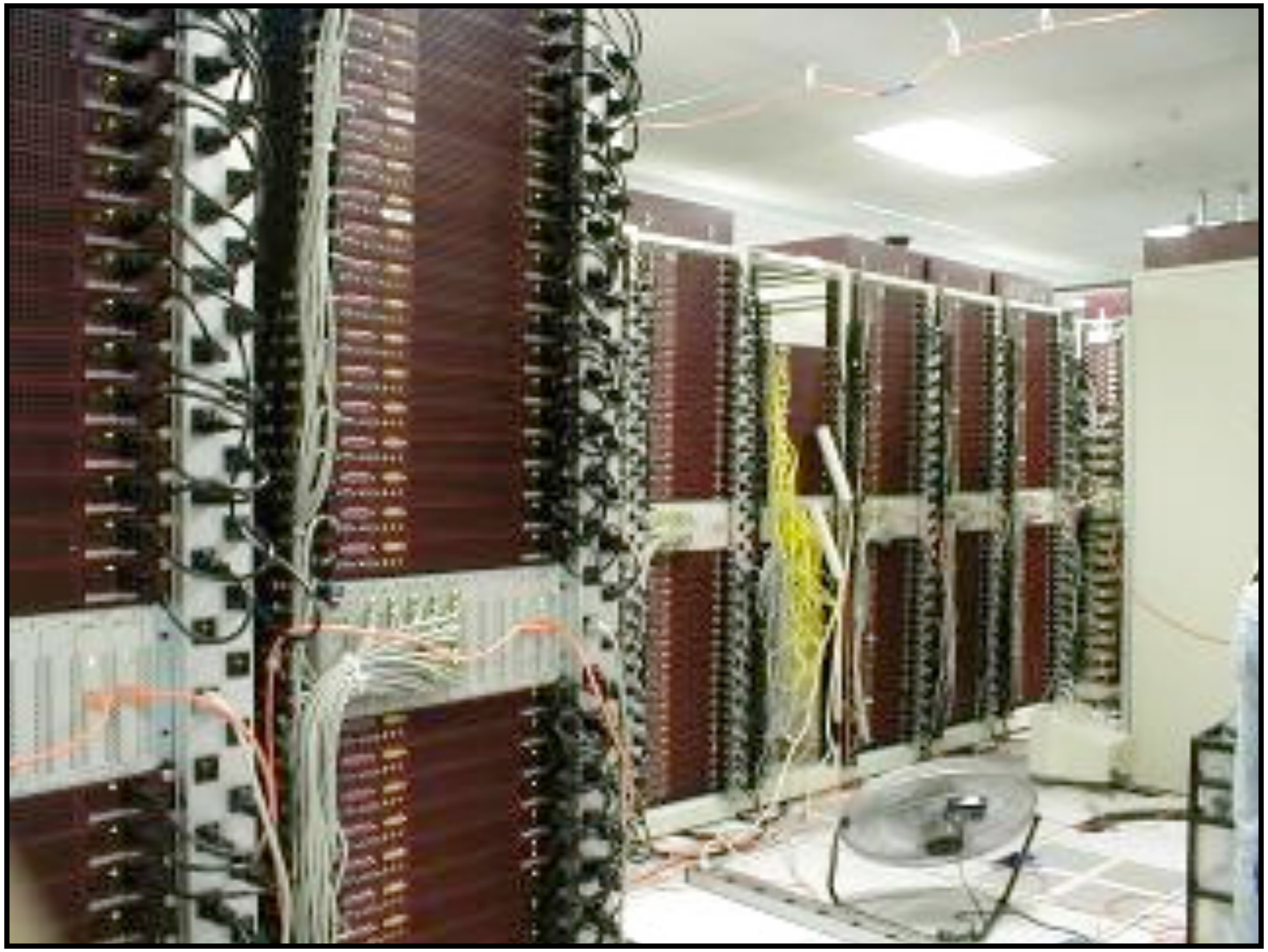
1. Copy new index to inner half of disk (while still serving old index)
2. Restart to use new index
3. Wipe old index
4. Re-copy new index to faster half of disk
5. Wipe first copy of new index
6. Inner half now free for building various performance improving data structures



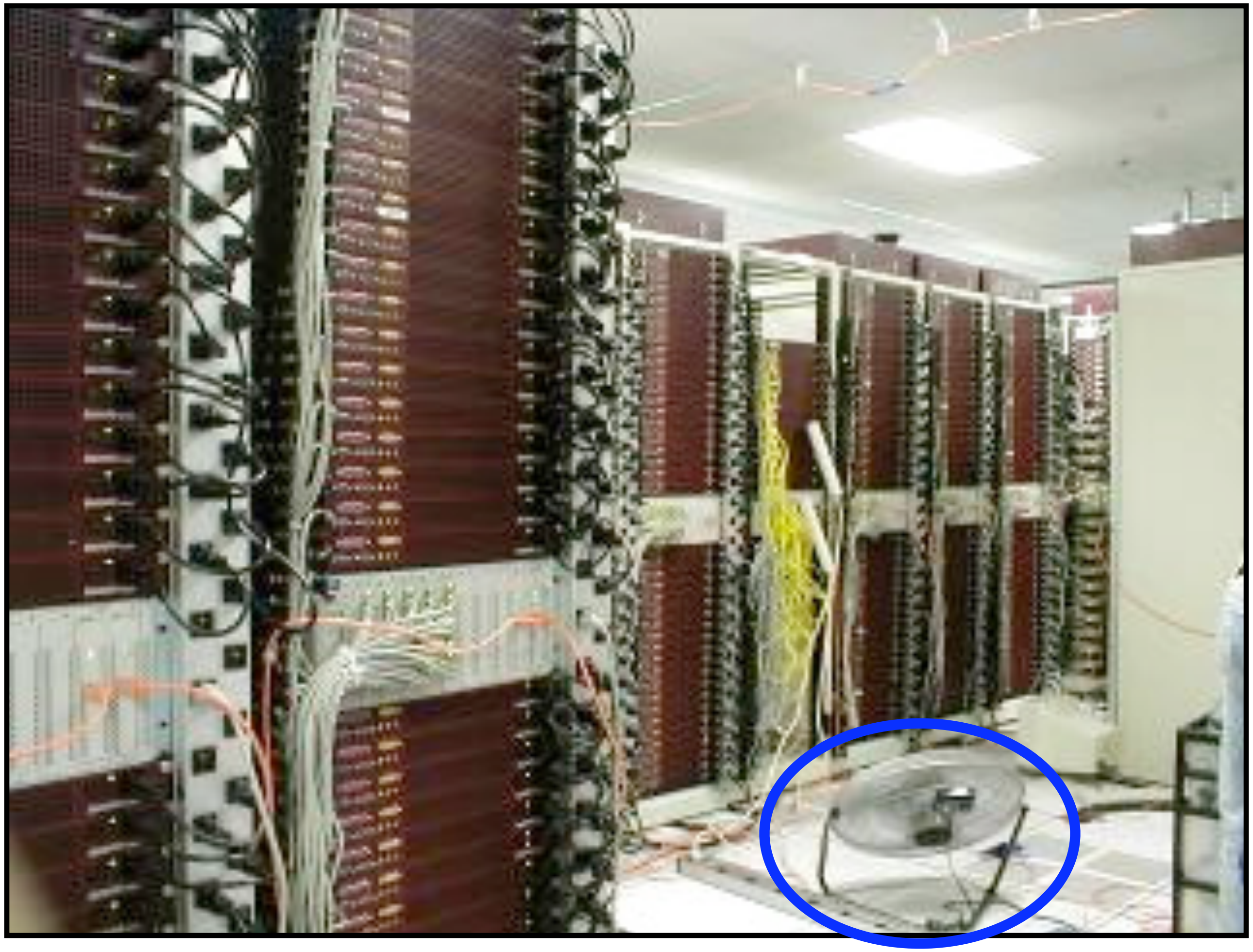
Pair cache: pre-intersected pairs of posting lists for commonly co-occurring query terms (e.g. “new” and “york”, or “barcelona” and “restaurants”)

Google Data Center (2000)

Google Data Center (2000)

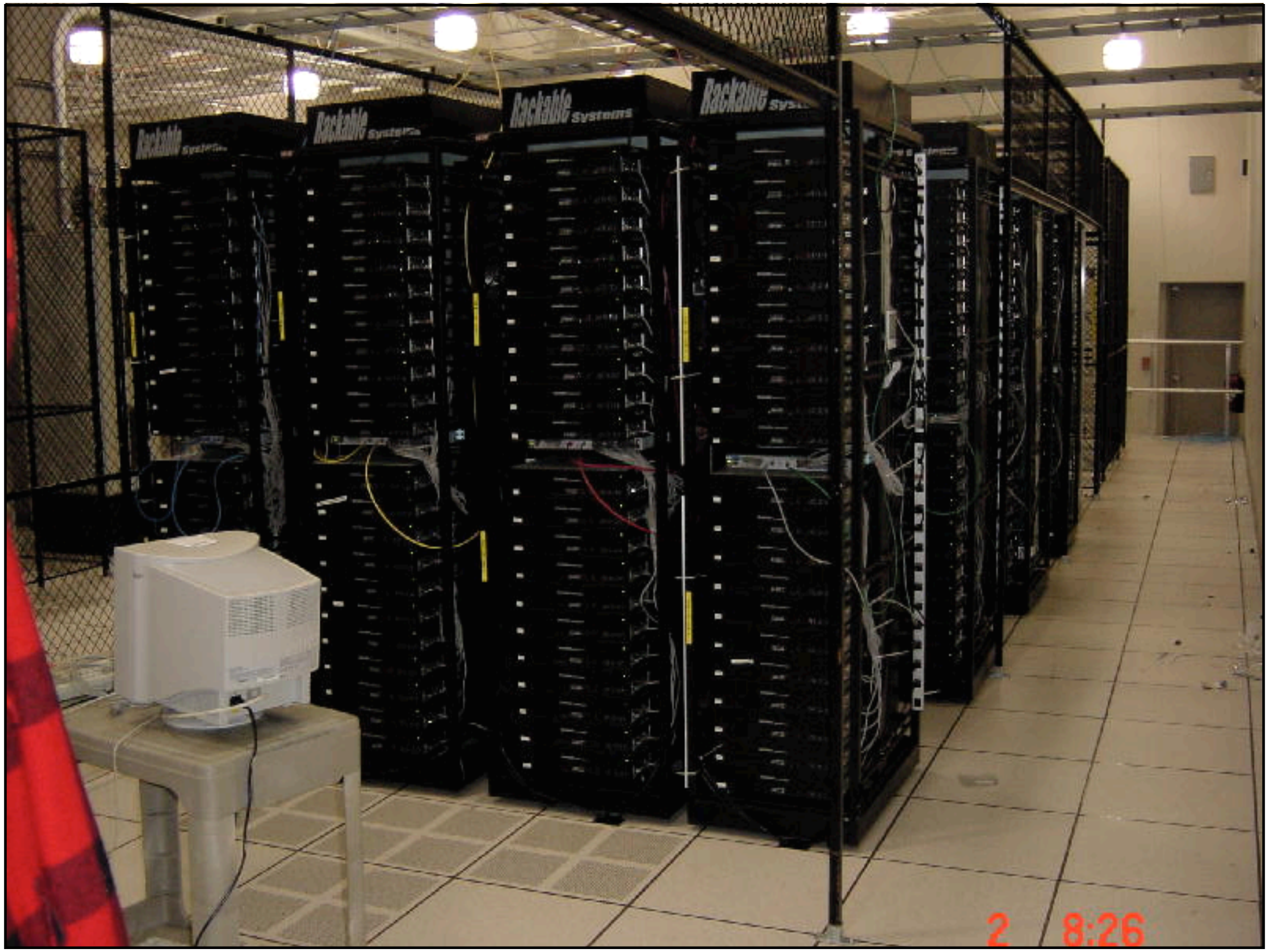


Google Data Center (2000)



Google (new data center 2001)

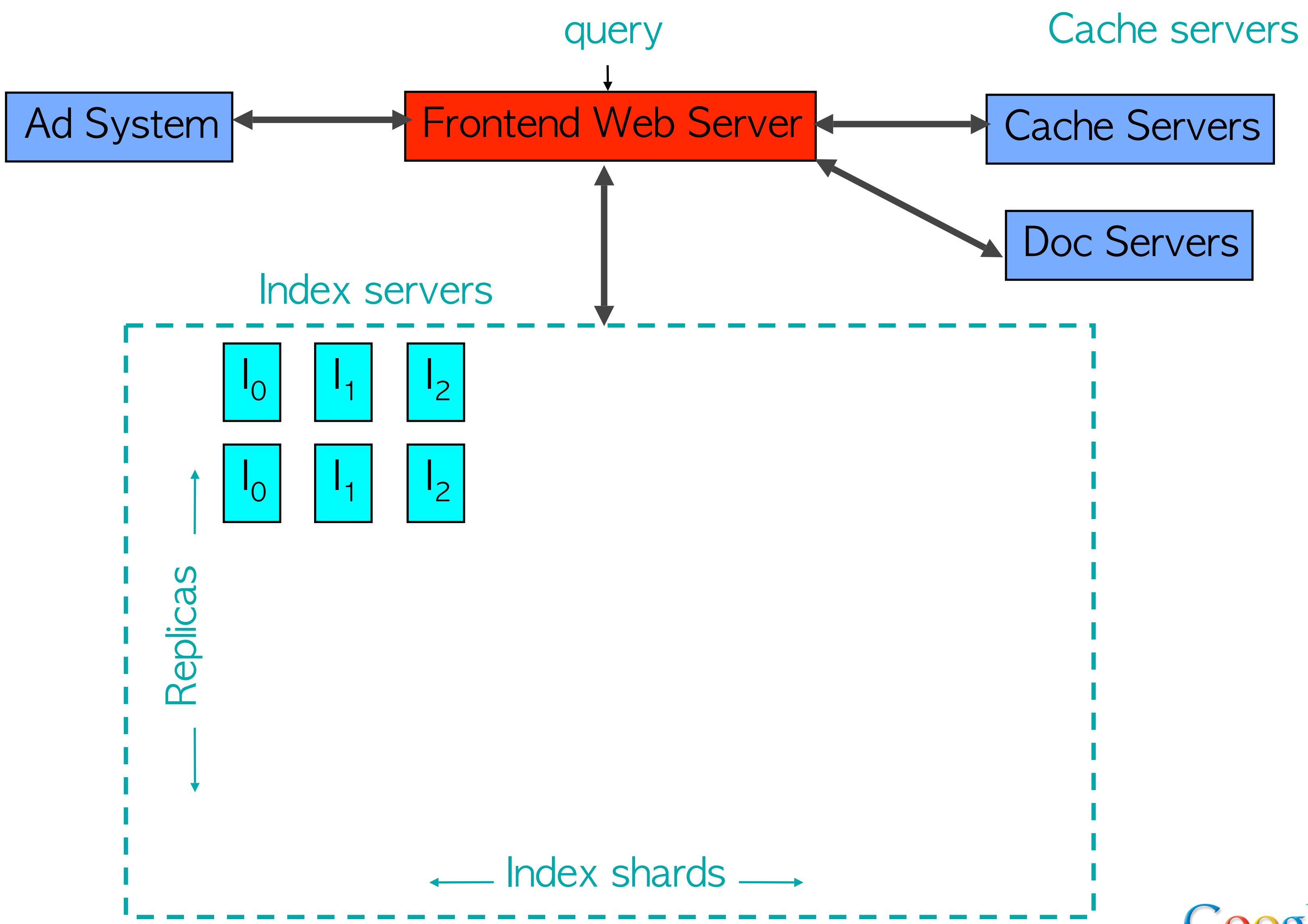
Google Data Center (3 days later)



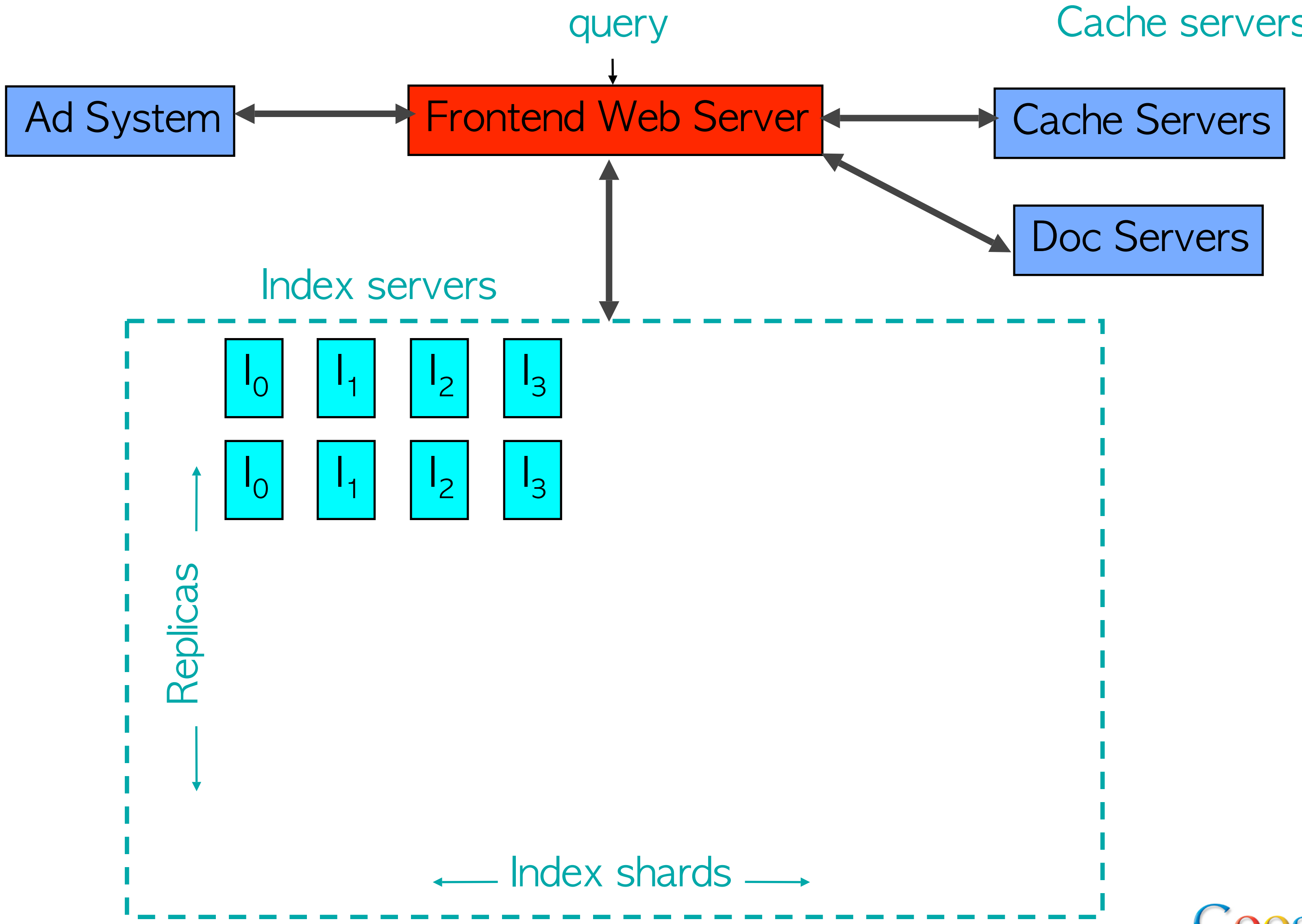
Increasing Index Size and Query Capacity

- Huge increases in index size in ’99, ’00, ’01, ...
  - From ~50M pages to more than 1000M pages
- At same time as huge traffic increases
  - ~20% growth per month in 1999, 2000, ...
  - ... plus major new partners (e.g. Yahoo in July 2000 doubled traffic overnight)
- Performance of index servers was paramount
  - Deploying more machines continuously, but...
  - Needed ~10-30% software-based improvement every month

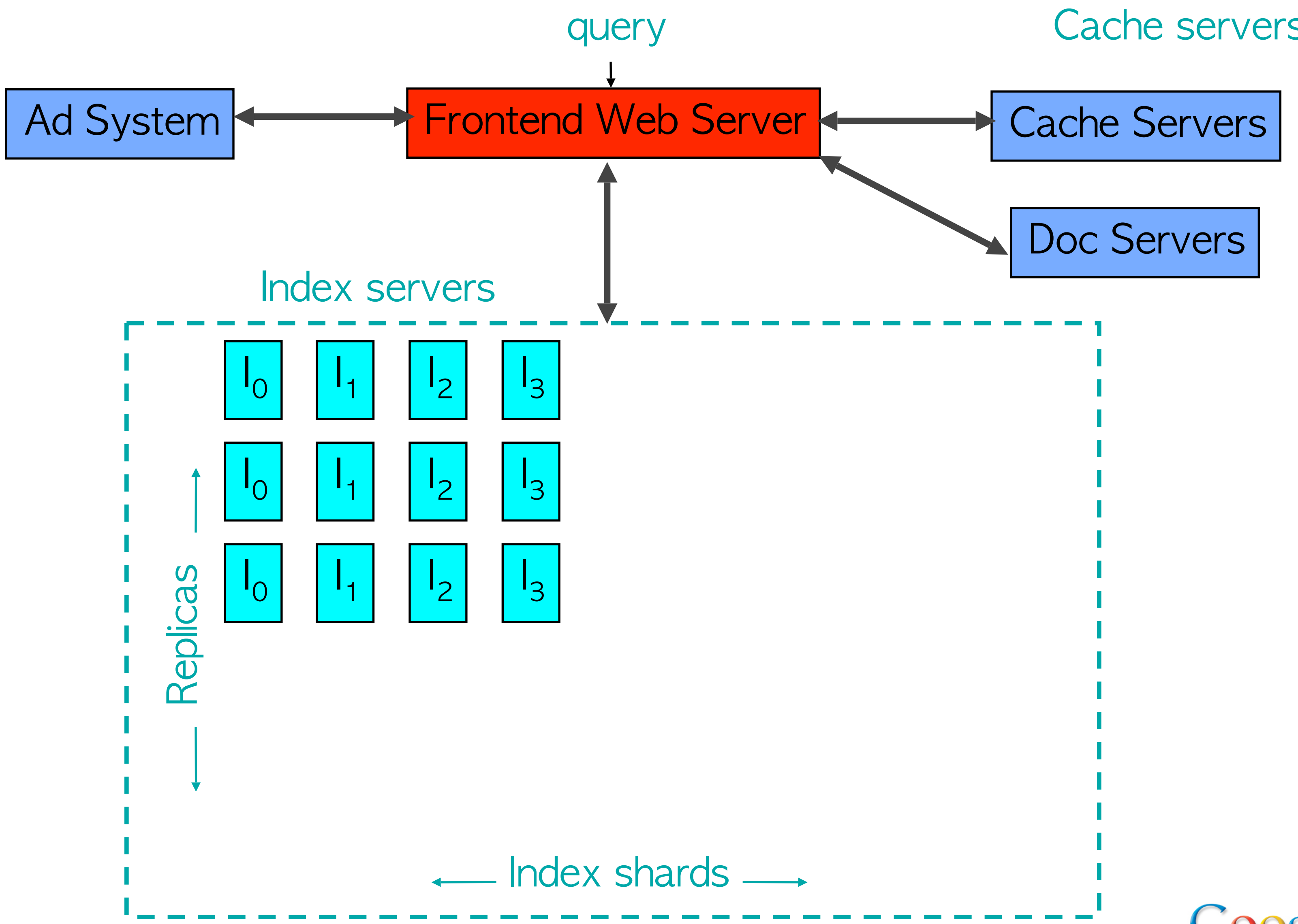
Dealing with Growth



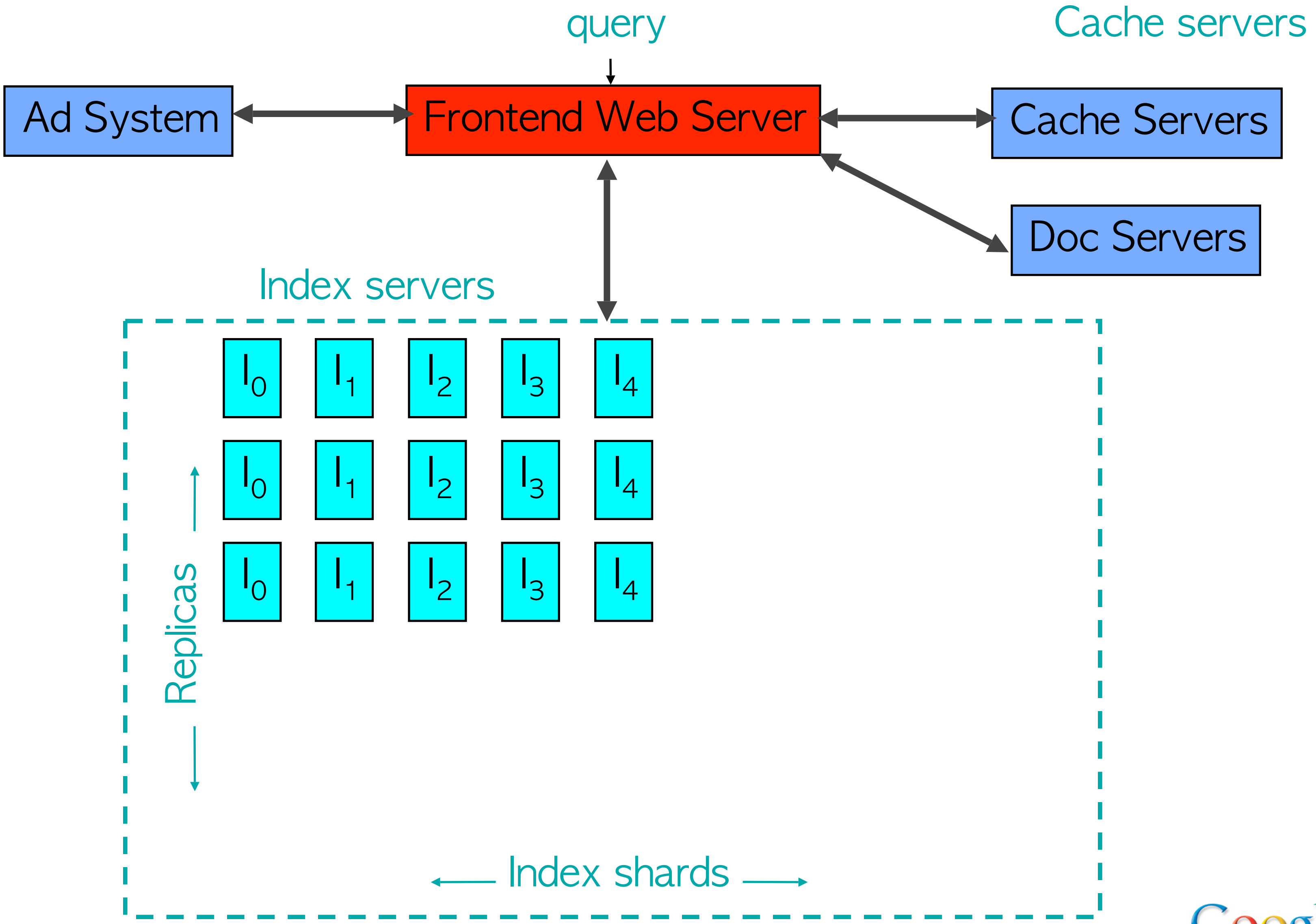
Dealing with Growth



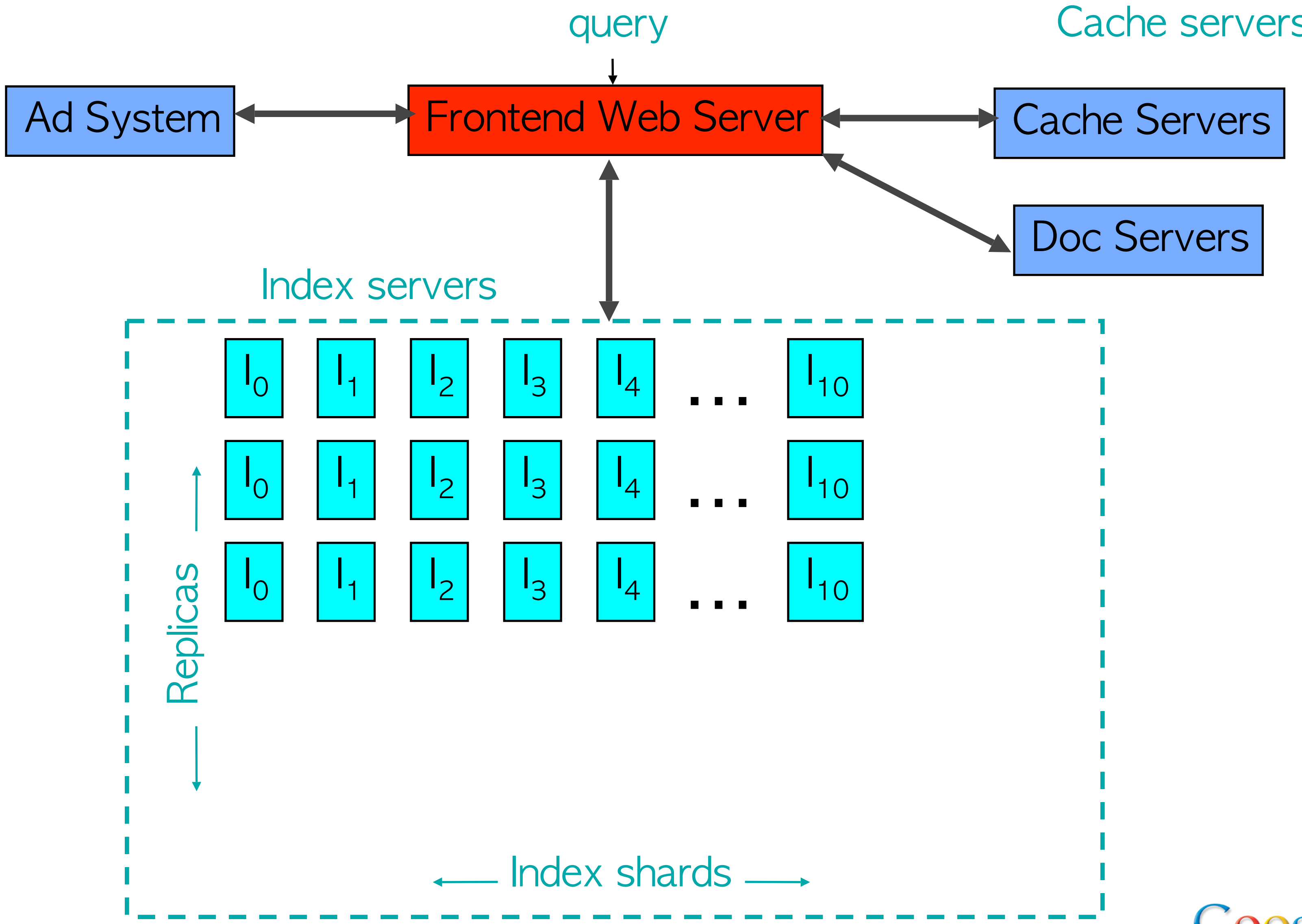
Dealing with Growth



Dealing with Growth

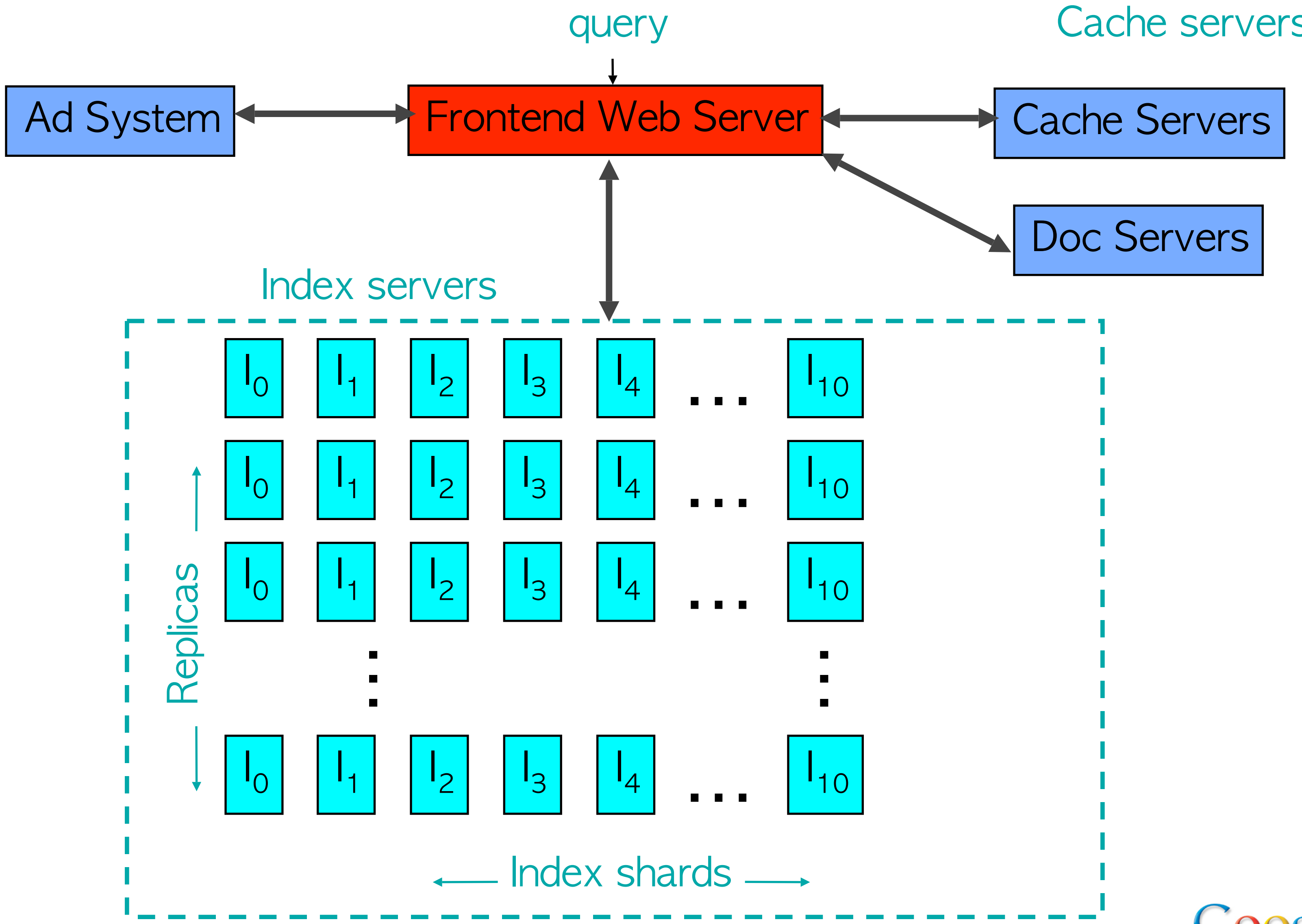


Dealing with Growth

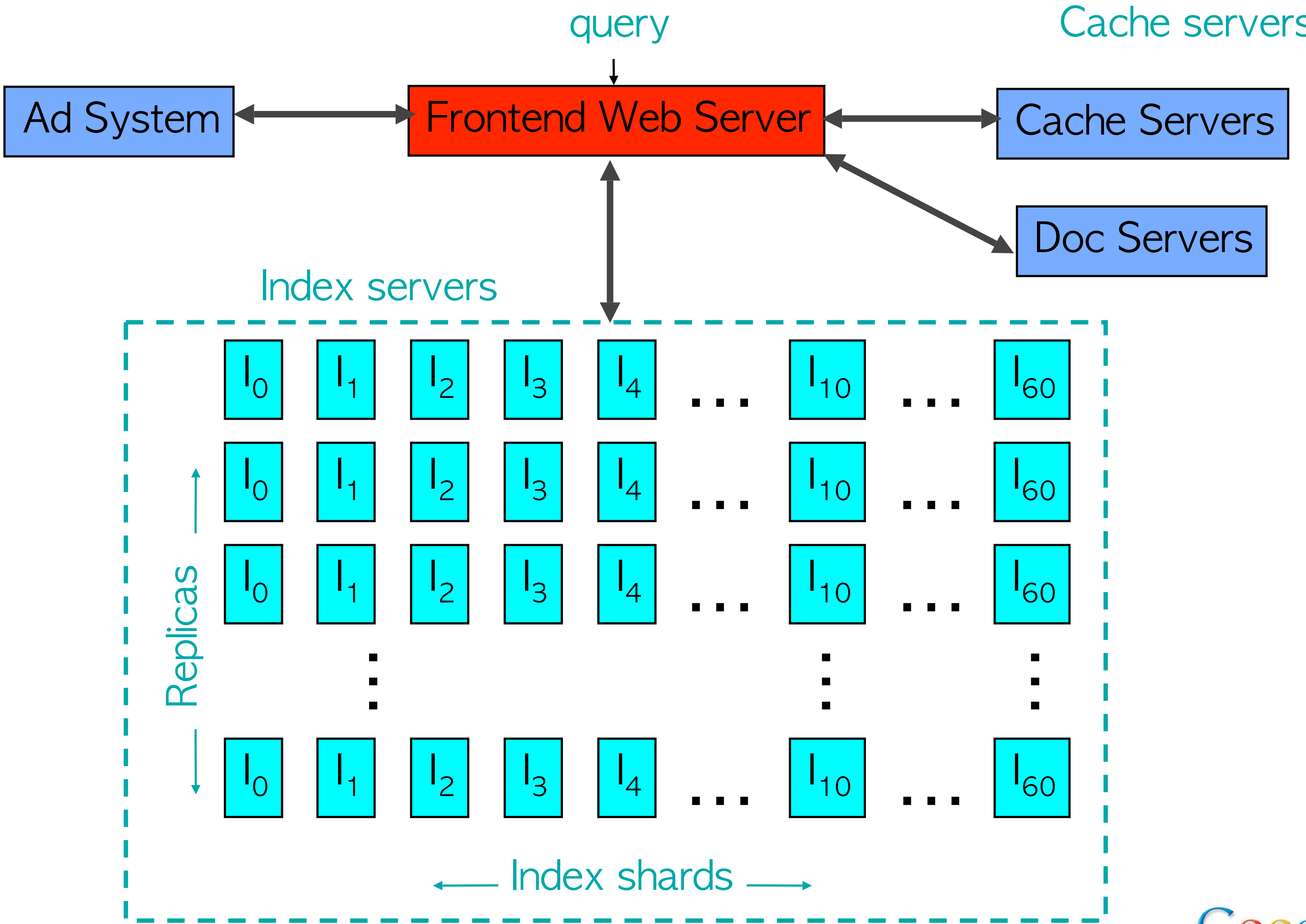


Google

Dealing with Growth



Dealing with Growth



Implications

- Shard response time influenced by:
  - # of disk seeks that must be done
  - amount of data to be read from disk

- Big performance improvements possible with:
  - better disk scheduling
  - improved index encoding

Index Encoding circa 1997-1999

- Original encoding ('97) was very simple:

WORD →	docid+nhits:32b	hit: 16b	hit: 16b	...	docid+nhits:32b	hit: 16b

  - hit: position plus attributes (font size, title, etc.)
  - Eventually added skip tables for large posting lists

- Simple, byte aligned format
  - cheap to decode, but not very compact
  - ... required lots of disk bandwidth

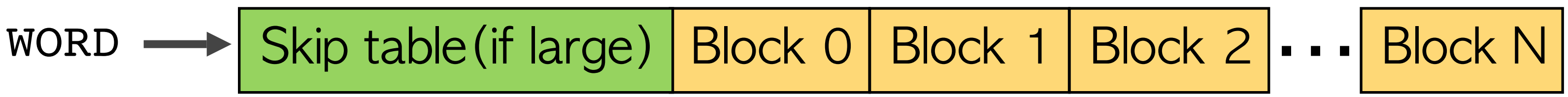
Encoding Techniques

- Bit-level encodings:
  - Unary: N '1's followed by a '0'
  - Gamma: $\log_2(N)$ in unary, then floor(log2(N)) bits
  - Rice_K: floor(N / 2^K) in unary, then N mod 2^K in K bits
    - special case of Golomb codes where base is power of 2
  - Huffman-Int: like Gamma, except $\log_2(N)$ is Huffman coded instead of encoded w/ Unary

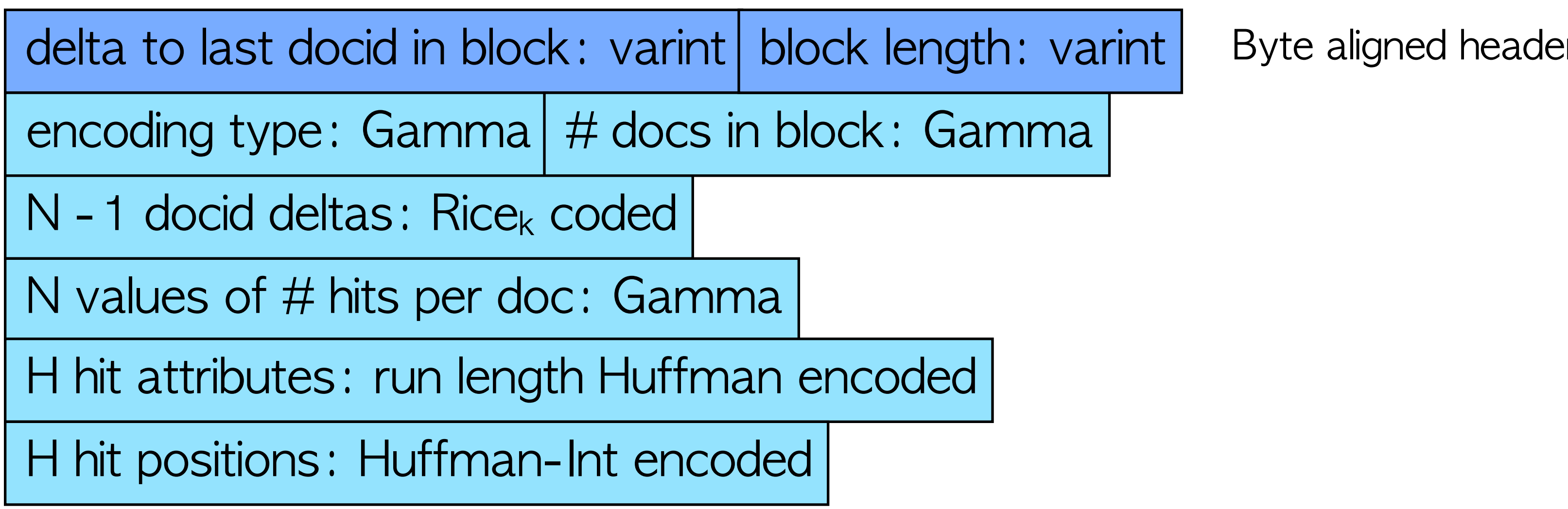
- Byte-aligned encodings:
  - varint: 7 bits per byte with a continuation bit
    - 0-127: 1 byte, 128-4095: 2 bytes, ...
  - ...

Block-Based Index Format

- Block-based, variable-len format reduced both space and CPU



Block format (with N documents and H hits):



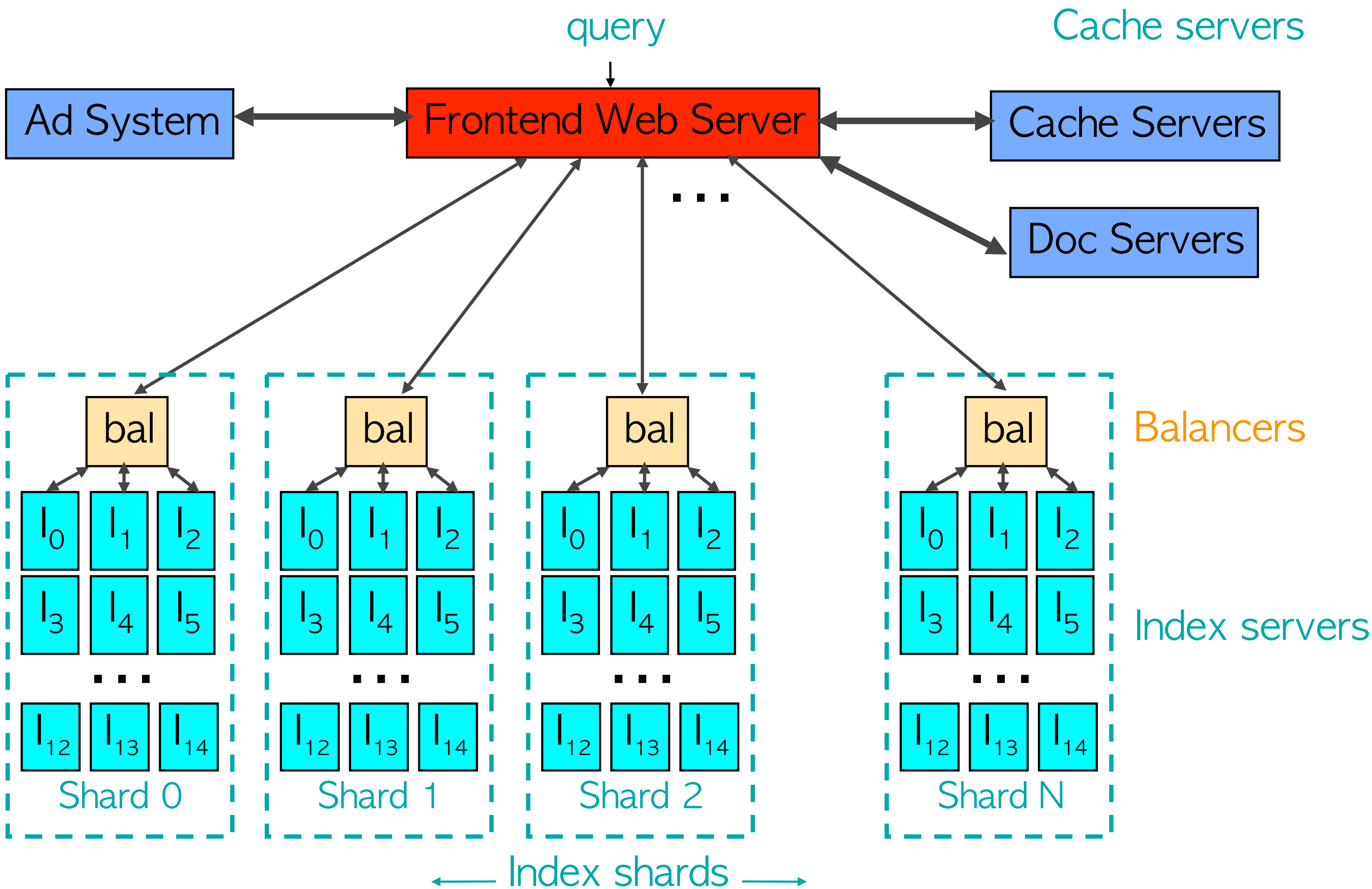
- Reduced index size by ~30%, plus much faster to decode

Implications of Ever-Wider Sharding

- Must add shards to keep response time low as index size increases
- ... but query cost increases with # of shards
  - typically >= 1 disk seek / shard / query term
  - even for very rare terms
- As # of replicas increases, total amount of memory available increases
  - Eventually, have enough memory to hold an **entire copy of the index in memory**
    - radically changes many design parameters

Google

Early 2001: In-Memory Index



In-Memory Indexing Systems

- Many positives:
  - big increase in throughput
  - big decrease in latency
    - especially at the tail: expensive queries that previously needed GBs of disk I/O became much faster

      `e.g. [ “circle of life” ]`

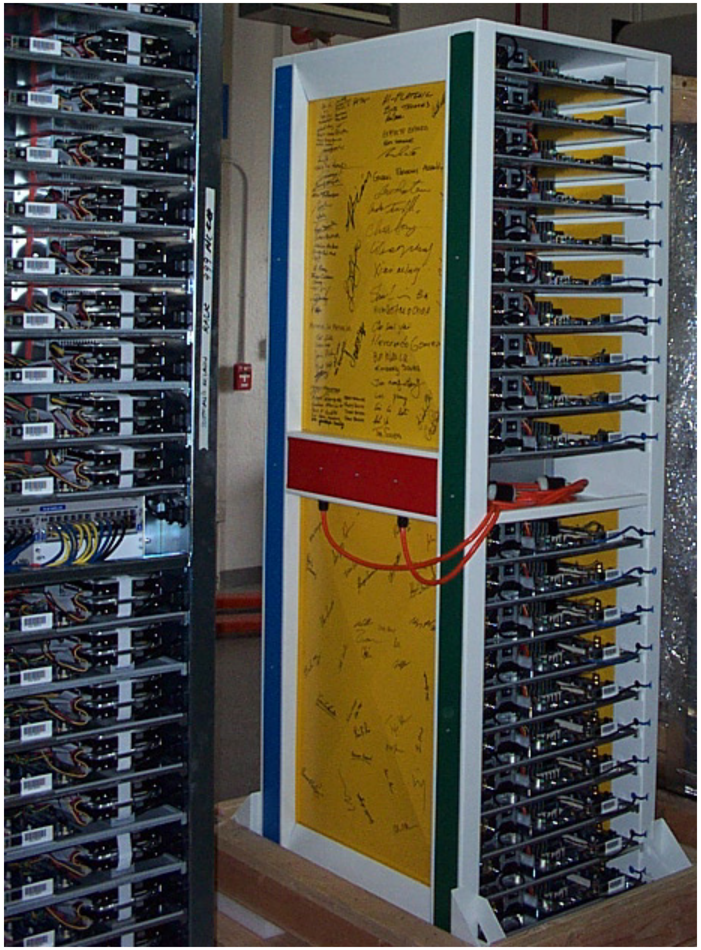
- Some issues:
  - Variance: touch 1000s of machines, not dozens
    - e.g. randomized cron jobs caused us trouble for a while
  - Availability: 1 or few replicas of each doc’s index data
    - Queries of death that kill all the backends at once: very bad
    - Availability of index data when machine failed (esp for important docs): replicate important docs

Google

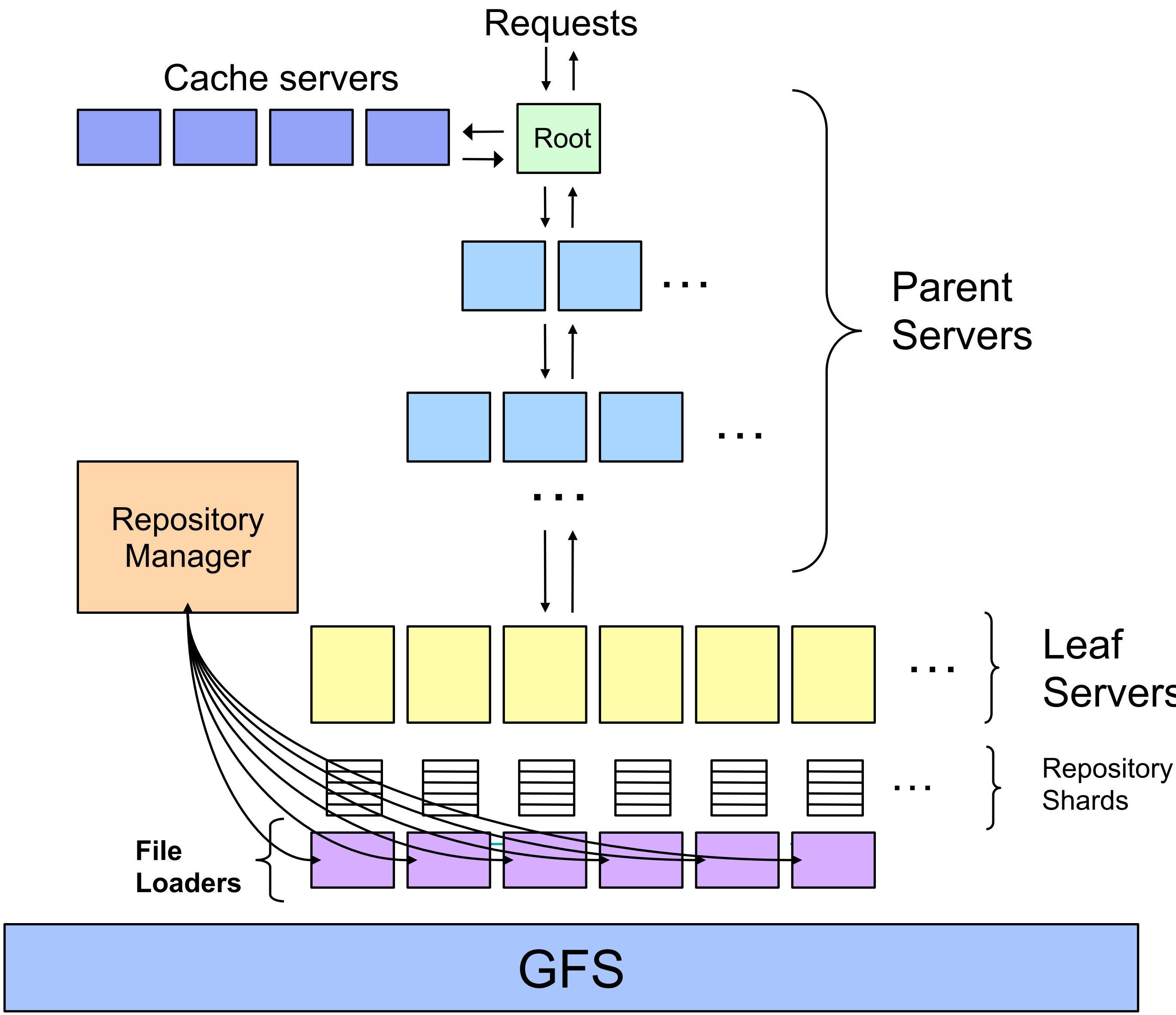
Larger-Scale Computing

Current Machines

- In-house rack design
- PC-class motherboards
- Low-end storage and networking hardware
- Linux
- + in-house software



Serving Design, 2004 edition

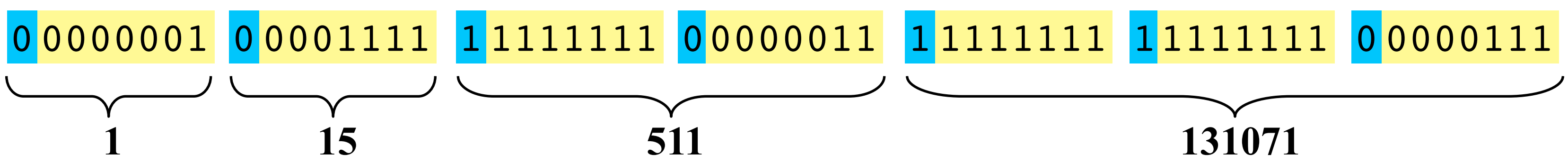


New Index Format

- Block index format used two-level scheme:
  - Each hit was encoded as (docid, word position in doc) pair
  - Docid deltas encoded with Rice encoding
  - Very good compression (originally designed for disk-based indices), but slow/CPU-intensive to decode
- New format: single flat position space
  - Data structures on side keep track of doc boundaries
  - Posting lists are just lists of delta-encoded positions
  - Need to be compact (can't afford 32 bit value per occurrence)
  - … but need to be very fast to decode

Byte-Aligned Variable-length Encodings

- Varint encoding:
  - 7 bits per byte with continuation bit
  - Con: Decoding requires lots of branches/shifts/masks



Little endian is used here (C/C++ on Intel CPUs). 511 = 0x01FF, but it will be actually saved as 0xFF01.

Byte-Aligned Variable-length Encodings

- Varint encoding:
  - 7 bits per byte with continuation bit
  - Con: Decoding requires lots of branches/shifts/masks

$$\underbrace{0\ 0000001}_{1}\quad \underbrace{0\ 0001111}_{15}\quad \underbrace{1\ 1111111\quad 0\ 0000011}_{511}\quad \underbrace{1\ 1111111\quad 1\ 1111111\quad 0\ 0000111}_{131071}$$

- Idea: Encode byte length as low 2 bits
  - Better: fewer branches, shifts, and masks
  - Con: Limited to 30-bit values, still some shifting to decode

$$\underbrace{00\ 000001}_{1}\quad \underbrace{00\ 001111}_{15}\quad \underbrace{01\ 111111\quad 00000011}_{511}\quad \underbrace{10\ 111111\quad 11111111\quad 00000111}_{131071}$$

weiw: should be "01 111111 00000111. This will give "00000111 111111" after dropping the 2 msbits and reversing the order of bytes. c.f., https://developers.google.com/protocol-buffers/docs/encoding

Group Varint Encoding

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix

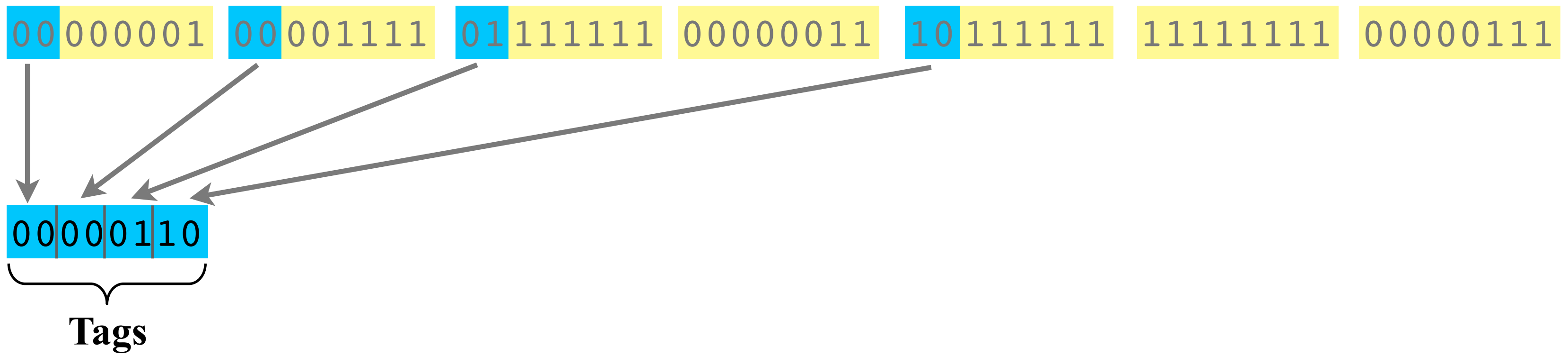
Group Varint Encoding

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix

`00000001` `00001111` `01111111` `00000011` `10111111` `11111111` `00000111`

Group Varint Encoding

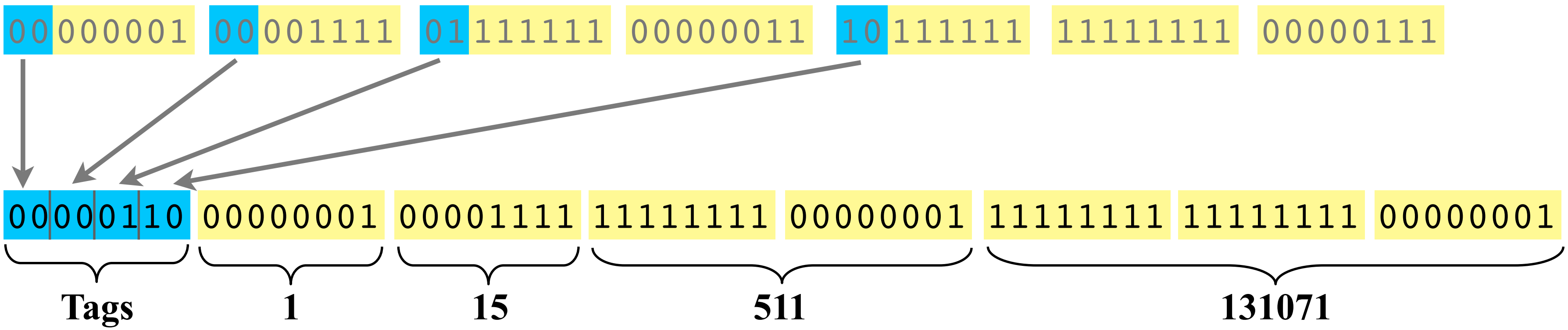
- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix



Google

Group Varint Encoding

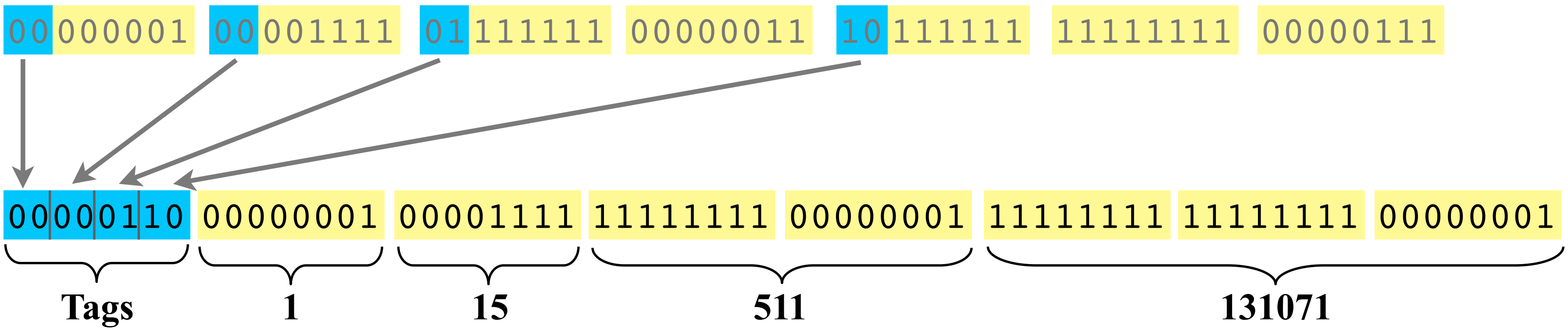
- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix



Google

Group Varint Encoding

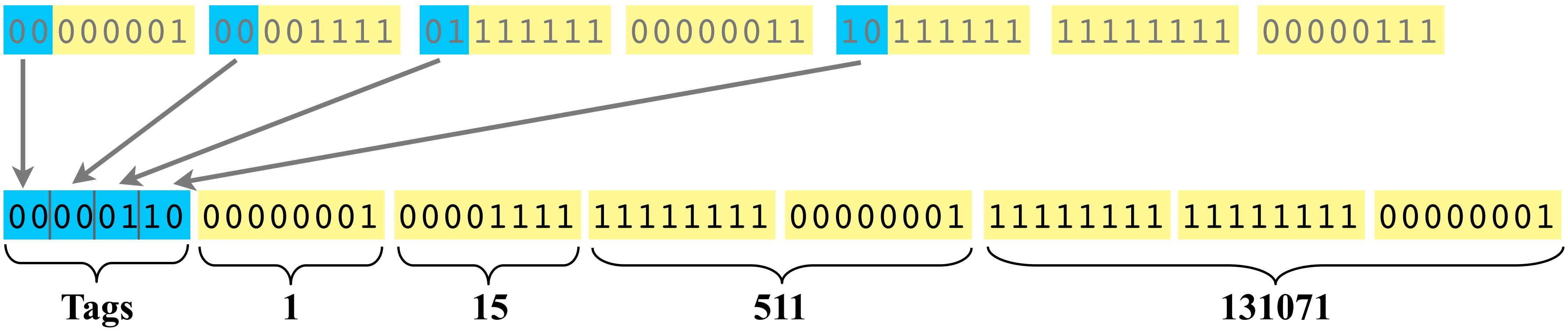
- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix



- Decode: Load prefix byte and use value to lookup in 256-entry table:

Group Varint Encoding

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix

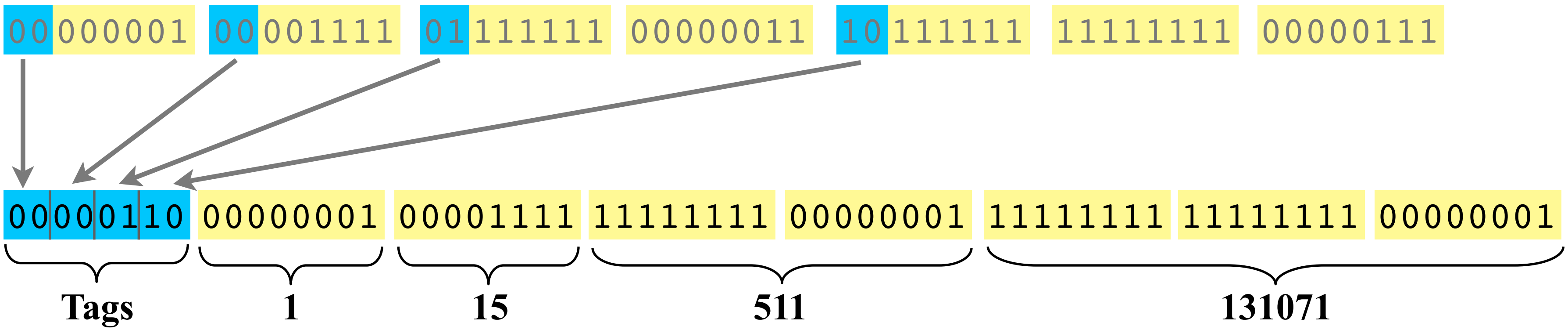


- Decode: Load prefix byte and use value to lookup in 256-entry table:

```
                    ...
00000110  →  Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffff
                    ...
```

Group Varint Encoding

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix

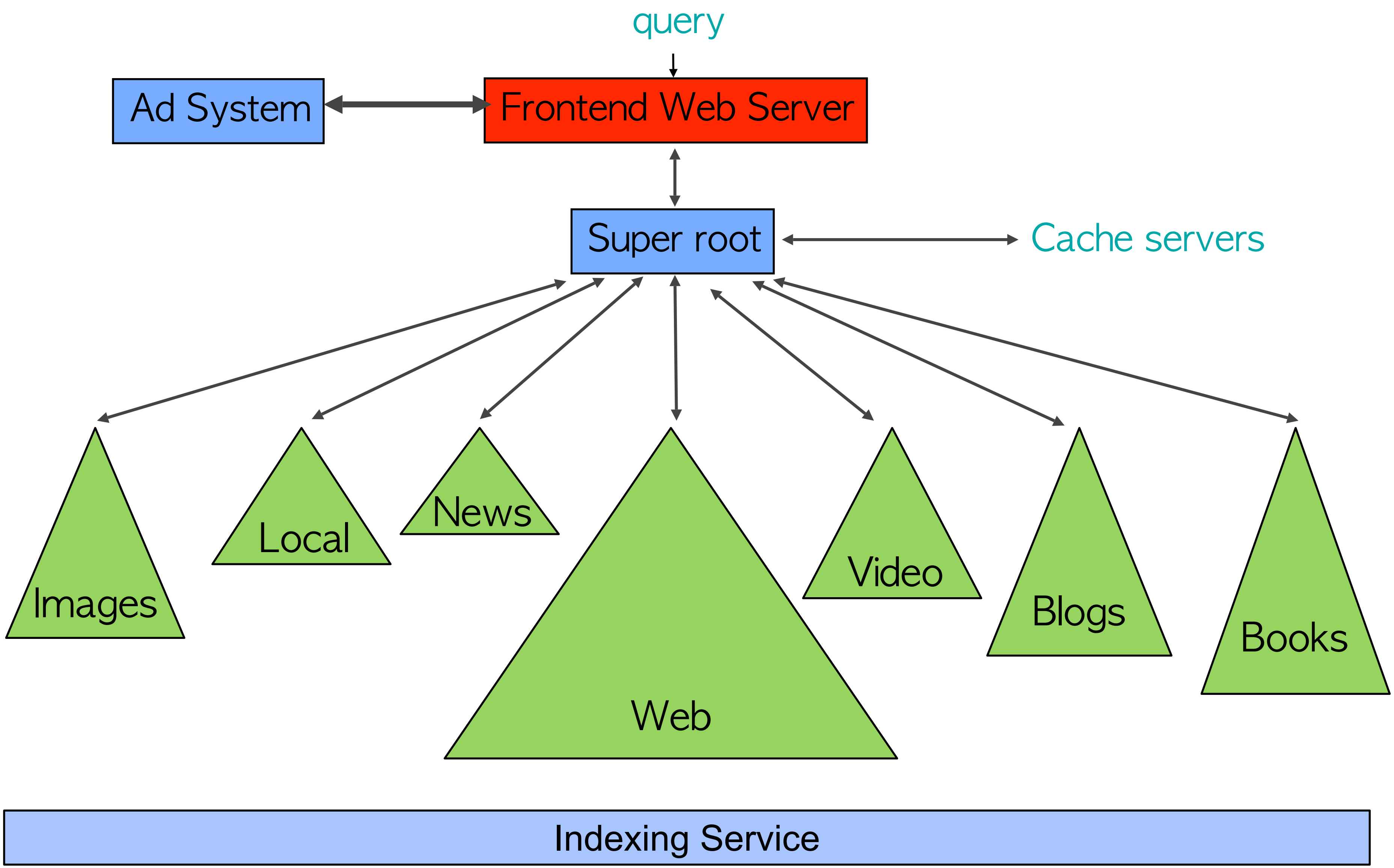


- Decode: Load prefix byte and use value to lookup in 256-entry table:

```
                   ...
00|00|01|10  →  Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffff
                   ...
```

- Much faster than alternatives:
  - 7-bit-per-byte varint: decode ~180M numbers/second
  - 30-bit Varint w/ 2-bit length: decode ~240M numbers/second
  - Group varint: decode ~400M numbers/second

2007: Universal Search



Index that? Just a minute!

- Low-latency crawling and indexing is tough
  - crawl heuristics: what pages should be crawled?
  - crawling system: need to crawl pages quickly
  - indexing system: depends on global data
    - PageRank, anchor text of pages that point to the page, etc.
    - must have online approximations for these global properties
  - serving system: must be prepared to accept updates while serving requests
    - very different data structures than batch update serving system

Flexibility & Experimentation in IR Systems

- Ease of experimentation hugely important
  - faster turnaround => more exps => more improvement
- Some experiments are easy
  - e.g. just weight existing data differently
- Others are more difficult to perform: need data not present in production index
  - Must be easy to generate and incorporate new data and use it in experiments

Infrastructure for Search Systems

- Several key pieces of infrastructure:
  - GFS: large-scale distributed file system
  - MapReduce: makes it easy to write/run large scale jobs
    - generate production index data more quickly
    - perform ad-hoc experiments more rapidly
    - ...
  - BigTable: semi-structured storage system
    - online, efficient access to per-document information at any time
    - multiple processes can update per-doc info asynchronously
    - critical for updating documents in minutes instead of hours

http://labs.google.com/papers/gfs.html

http://labs.google.com/papers/mapreduce.html

http://labs.google.com/papers/bigtable.html

Experimental Cycle, Part 1

- Start with new ranking idea

- Must be easy to run experiments, and to do so quickly:
  - Use tools like MapReduce, BigTable, to generate data...
  - Initially, run off-line experiment & examine effects
    - ...on human-rated query sets of various kinds
    - ...on random queries, to look at changes to existing ranking

  - Latency and throughput of this prototype don't matter

- ...iterate, based on results of experiments ...

Experimental Cycle, Part 2

- Once off-line experiments look promising, want to run live experiment
  - Experiment on tiny sliver of user traffic
  - Random sample, usually
    - but sometimes a sample of specific class of queries
      - e.g. English queries, or queries with place names, etc.

- For this, throughput not important, but latency matters
  - Experimental framework must operate at close to production latencies!

Experiment Looks Good: Now What?

- Launch!

- Performance tuning/reimplementation to make feasible at full load
  - e.g. precompute data rather than computing at runtime
  - e.g. approximate if "good enough" but much cheaper

- Rollout process important:
  - Continuously make quality vs. cost tradeoffs
  - Rapid rollouts at odds with low latency and site stability
    - Need good working relationships between search quality and groups chartered to make things fast and stable

Future Directions & Challenges

- A few closing thoughts on interesting directions...

Cross-Language Information Retrieval

- Translate all the world's documents to all the world's languages
  - increases index size substantially
  - computationally expensive
  - ... but huge benefits if done well

- Challenges:
  - continuously improving translation quality
  - large-scale systems work to deal with larger and more complex language models
    - to translate one sentence ⇒ ~1M lookups in multi-TB model

ACLs in Information Retrieval Systems

- Retrieval systems with mix of private, semi-private, widely shared and public documents
  - e.g. e-mail vs. shared doc among 10 people vs. messages in group with 100,000 members vs. public web pages
- Challenge: building retrieval systems that efficiently deal with ACLs that vary widely in size
  - best solution for doc shared with 10 people is different than for doc shared with the world
  - sharing patterns of a document might change over time

Automatic Construction of Efficient IR Systems

- Currently use several retrieval systems
  - e.g. one system for sub-second update latencies, one for very large # of documents but daily updates, ...
  - common interfaces, but very different implementations primarily for efficiency
  - works well, but lots of effort to build, maintain and extend different systems

- Challenge: can we have a single parameterizable system that automatically constructs efficient retrieval system based on these parameters?

Information Extraction from Semi-structured Data

- Data with clearly labelled semantic meaning is a tiny fraction of all the data in the world
- But there's lots semi-structured data
  - books & web pages with tables, data behind forms, ...

- Challenge: algorithms/techniques for improved extraction of structured information from unstructured/semi-structured sources
  - noisy data, but lots of redundancy
  - want to be able to correlate/combine/aggregate info from different sources

In Conclusion...

- Designing and building large-scale retrieval systems is a challenging, fun endeavor
  - new problems require continuous evolution
  - work benefits many users
  - new retrieval techniques often require new systems

- Thanks for your attention!

Thanks! Questions...?

- Further reading:

Ghemawat, Gobioff, & Leung. *Google File System*, SOSP 2003.

Barroso, Dean, & Hölzle. *Web Search for a Planet: The Google Cluster Architecture*, IEEE Micro, 2003.

Dean & Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004.

Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber. *Bigtable: A Distributed Storage System for Structured Data*, OSDI 2006.

Brants, Popat, Xu, Och, & Dean. *Large Language Models in Machine Translation*, EMNLP 2007.

- These and many more available at:

  http://labs.google.com/papers.html

Google