Performance Engineering of Large-Scale Systems: Lessons Learned in Building Search and Infrastructure Software

Jeff Dean
Google Senior Fellow
jeff@google.com
Plan

- Walk through several generations of search systems and infrastructure systems
- Cover some common patterns and general rules of thumb for similar systems
 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. today

<table>
<thead>
<tr>
<th>Aspect</th>
<th>1999</th>
<th>Today</th>
</tr>
</thead>
<tbody>
<tr>
<td># docs: ~70M to many tens of billions</td>
<td></td>
<td>~1000X</td>
</tr>
<tr>
<td>queries processed/day:</td>
<td></td>
<td>~1000X</td>
</tr>
<tr>
<td>per doc info in index:</td>
<td></td>
<td>~5X</td>
</tr>
<tr>
<td>update latency: months to minutes</td>
<td></td>
<td>~10000X</td>
</tr>
<tr>
<td>avg. query latency: &lt;1s to &lt;0.2s</td>
<td></td>
<td>~5X</td>
</tr>
</tbody>
</table>

- More machines * faster machines: ~5000X
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 first 10 years of Google
  – often rolled out without users realizing we’ve made major changes
The Crayon Chart
It started with the question, “What factors increase Google usage?” I had historical traffic stats, but I didn’t have the history of what had happened before I joined. I wanted to encourage people to annotate a timeline, and I wanted to keep it informal, so that people didn’t worry about “messing it up.” So I drew it in crayon.

It worked. People marked off when they joined the company, when they launched a product, when we hit a milestone. For several years, until we became a big company that wouldn’t fit on a sheet of paper, it was a record of our history.

-Lucas Pereira
“Google” Circa 1997 (google.stanford.edu)
Research Project, circa 1997

Frontend Web Server

- Index servers: $I_0, I_1, I_2, \ldots, I_N$
- Doc servers: $D_0, D_1, \ldots, D_M$

Query flow:
- Frontend Web Server queries Index servers and Doc servers.
- Index shards: $I_0, I_1, I_2, \ldots, I_N$ are connected to the Index servers.
- Doc shards: $D_0, D_1, \ldots, D_M$ are connected to the Doc servers.
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

  In our computing environment, **by doc** makes more sense

• **By word**: shard has subset of words for all docs
  – pro: K word query $\Rightarrow$ 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
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 $O(\# \text{ queries} \times \# \text{ docs in index})$
- Doc Servers
  - given (docid, query) generate (title, snippet)
  - map from docid to full text of docs on disk
  - also partitioned by docid
  - cost is $O(\# \text{ queries})$
“Corkboards” (1999)
Serving System, circa 1999

Ad System → Frontend Web Server

Frontend Web Server

Query

Cache servers

Index servers

Doc servers

Index shards

Doc shards

Replicas

Replicas
Indexing (circa 1998-1999)

• Original 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

Nov99 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 (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

Ad System → Frontend Web Server → Cache Servers

Query flow:
- Index servers (I₀, I₁, I₂, I₃, I₄, ..., I₁₀, ..., I₆₀)
- Cache servers

Replicas:
- Index shards

Diagram by Google
Index Encoding circa 1997-1999

• Original encoding (’97) was very simple:

  - 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 \( \text{floor}(\log_2(N)) \) bits
  – **Rice}_K**: \( \text{floor}(N / 2^K) \) 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):

- delta to last docid in block: varint
- block length: varint
- encoding type: Gamma
- # docs in block: Gamma
- $N-1$ docid deltas: Rice$_k$ coded
- $N$ values of # hits per doc: Gamma
- H hit attributes: run length Huffman encoded
- H hit positions: Huffman-Int encoded

- 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
Early 2001: In-Memory Index

Ad System → Frontend Web Server → Index servers → Cache servers

Balancers

Index servers

Shard 0: I₀, I₁, I₂, I₃, I₄, I₅, I₁₂, I₁₃, I₁₄
Shard 1: I₀, I₁, I₂, I₃, I₄, I₅, I₁₂, I₁₃, I₁₄
Shard 2: I₀, I₁, I₂, I₃, I₄, I₅, I₁₂, I₁₃, I₁₄
Shard N: I₀, I₁, I₂, I₃, I₄, I₅, I₁₂, I₁₃, I₁₄

Index shards
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 and cheaper
      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
Data Independent Failures
Canary Requests

query
Serving Design, 2004 edition

Repository Manager

Requests

Cache servers

Root

Parent Servers

Leaf Servers

Repository Shards

GFS

File Loaders
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

  ![Varint Encoding Example]

• **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

  ![Byte-Length Encoding Example]
Group Varint Encoding

• Idea: encode groups of 4 32-bit values in 5-17 bytes
  – Pull out 4 2-bit binary lengths into single byte prefix

0000110 0000001 00001111 11111111 00000111
0000110 0000001 00001111 11111111 00000111
00000001 00001111 11111111 00000111 11111111 00000111

• Decode: Load prefix byte and use value to lookup in 256-entry table:
  ... 00000110 \rightarrow Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffffff ...

• 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
Some Commonly Used Systems Infrastructure at Google

- **GFS & Colossus** (next gen GFS)
  - cluster-level file system (distributed across thousands of nodes)
- **MapReduce**
  - programming model and implementation for large-scale computation
- **Bigtable**
  - distributed semi-structured storage system
  - adaptively spreads data across thousands of nodes
  - can loosely couple many Bigtable setups in different clusters with eventual consistency replication
- **Spanner**
  - geographically distributed worldwide structured storage system
  - adaptively spreads data across thousands of nodes in each of dozens of datacenters
  - writes are strongly consistent using Paxos
  - reads can be either strongly consistent or weakly consistent
Pattern: Combine Multiple Implementations

• Example: Google web search system wants all of these:
  – freshness (update documents in ~1 second)
  – massive capacity (10000s of requests per second)
  – high quality retrieval (lots of information about each document)
  – massive size (billions of documents)

• Very difficult to accomplish in single implementation

• Partition problem into several subproblems with different engineering tradeoffs. E.g.
  – realtime system: few docs, ok to pay lots of $$$$/doc
  – base system: high # of docs, optimized for low $/doc
  – realtime+base: high # of docs, fresh, low $/doc
Designing Efficient Systems

Given a basic problem definition, how do you choose "best" solution?

- Best could be simplest, highest performance, easiest to extend, etc.

Important skill: ability to estimate performance of a system design
- without actually having to build it!
# Numbers Everyone Should Know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K w/cheap compression algorithm</td>
<td>3,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
Back of the Envelope Calculations

How long to generate image results page (30 thumbnails)?

Design 1: Read serially, thumbnail 256K images on the fly

\[ 30 \text{ seeks} \times 10 \text{ ms/seek} + 30 \times 256K / 30 \text{ MB/s} = 560 \text{ ms} \]

Design 2: Issue reads in parallel:

\[ 10 \text{ ms/seek} + 256K \text{ read} / 30 \text{ MB/s} = 18 \text{ ms} \]

(Ignores variance, so really more like 30-60 ms, probably)

Lots of variations:

- caching (single images? whole sets of thumbnails?)
- pre-computing thumbnails
- ...

Back of the envelope helps identify most promising…
Know Your Basic Building Blocks

Core language libraries, basic data structures, protocol buffers, GFS, BigTable, indexing systems, MapReduce, …

Not just their interfaces, but understand their implementations (at least at a high level)

If you don’t know what’s going on, you can’t do decent back-of-the-envelope calculations!

• Corollary: implementations with unpredictable 1000X variations in performance are not very helpful if latency or throughput matters
  – e.g. VM paging
Design for Growth

Try to anticipate how requirements will evolve
  keep likely features in mind as you design base system

Don’t design to scale infinitely:
  ~5X - 50X growth good to consider
  >100X probably requires rethink and rewrite
Pattern: Single Master, 1000s of Workers

- Master orchestrates global operation of system
  - load balancing, assignment of work, reassignment when machines fail, etc.
  - ... but client interaction with master is fairly minimal

- Examples:
  - GFS, BigTable, MapReduce, Transfer Service, cluster scheduling system, ...

![Diagram showing the single master with multiple workers and replicas connected to clients and miscellaneous servers. Masters, replicas, and workers are represented with different shapes and colors.]
Pattern: Single Master, 1000s of Workers (cont)

- Often: **hot standby of master waiting to take over**
- Always: **bulk of data transfer directly between clients and workers**

- Pro:
  - simpler to reason about state of system with centralized master

- Caveats:
  - careful design required to keep master out of common case ops
  - scales to 1000s of workers, but not 100,000s of workers
Pattern: Tree Distribution of Requests

- Problem: Single machine sending 1000s of RPCs overloads NIC on machine when handling replies
  - wide fan in causes TCP drops/retransmits, significant latency
  - CPU becomes bottleneck on single machine
Pattern: Tree Distribution of Requests

- Solution: Use tree distribution of requests/responses
  - fan in at root is smaller
  - cost of processing leaf responses spread across many parents
- Most effective when parent processing can trim/combine leaf data
  - can co-locate parents "closer" in network topology to leaves
Pattern: Backup Requests to Minimize Latency

• Problem: variance high when requests go to 1000s of machines
  – last few machines stretch out latency tail

• Often, multiple replicas can handle same kind of request
• When few tasks remaining, send backup requests to other replicas
• Whichever duplicate request finishes first wins
  – useful when variance is unrelated to specifics of request
  – increases overall load by a tiny percentage
  – decreases latency tail significantly

• Examples:
  – MapReduce backup tasks (granularity: many seconds)
  – various query serving systems (granularity: milliseconds)
Pattern: Multiple Smaller Units per Machine

• Problems:
  – want to minimize recovery time when machine crashes
  – want to do fine-grained load balancing

• Having each machine manage 1 unit of work is inflexible
  – slow recovery: new replica must recover data that is $O(\text{machine state})$ in size
  – load balancing much harder
Pattern: Multiple Smaller Units per Machine

- Have each machine manage many smaller units of work/data
  - typical: ~10-100 units/machine
  - allows fine grained load balancing (shed or add one unit)
  - fast recovery from failure (N machines each pick up 1 unit)

- Examples:
  - map and reduce tasks, GFS chunks, Bigtable tablets, query serving system index shards
Many machines each recover one or a few partition – e.g. BigTable tablets, GFS chunks, query serving shards
Pattern: Elastic Systems

• Problem: Planning for exact peak load is hard
  – overcapacity: wasted resources
  – undercapacity: meltdown

• Design system to adapt:
  – automatically shrink capacity during idle period
  – automatically grow capacity as load grows

• Make system resilient to overload:
  – do something reasonable even up to 2X planned capacity
    • e.g. shrink size of index searched, back off to less CPU intensive algorithms, drop spelling correction tip, etc.
  – more aggressive load balancing when imbalance more severe
Selective Replication

- Find heavily used items and make more replicas
  - can be static or dynamic

- Example: Query serving system
  - static: more replicas of important docs
  - dynamic: more replicas of Chinese documents as Chinese query load increases
Monitoring and Debugging

• Questions you might want to ask:
  – did this change I rolled out last week affect # of errors / request?
  – why are my tasks using so much memory?
  – where is CPU time being spent in my application?
  – what kinds of requests are being handled by my service?
  – why are some requests very slow?

• Important to have enough visibility into systems to answer these kinds of questions
Add Sufficient Monitoring/Status/Debugging Hooks

All our servers:

- Export HTML-based status pages for easy diagnosis
- Export a collection of key-value pairs via a standard interface
  - monitoring systems periodically collect this from running servers
- RPC subsystem collects sample of all requests, all error requests, all requests >0.0s, >0.05s, >0.1s, >0.5s, >1s, etc.

- Support low-overhead online profiling
  - cpu profiling
  - memory profiling
  - lock contention profiling

If your system is slow or misbehaving, can you figure out why?
Exported Variables

• Special URL on every Google server

```
rpc-server-count-minute        11412
rpc-server-count             502450983
rpc-server-arg-bytes-minute  8039419
rpc-server-arg-bytes         372908296166
rpc-server-rpc-errors-minute 0
rpc-server-rpc-errors        0
rpc-server-app-errors-minute  8
rpc-server-app-errors        2357783
uptime-in-ms                 679532636
build-timestamp-as-int       1343415737
build-timestamp             "Built on Jul 27 2012 12:02:17 (1343415737)"
...```

• On top of this, we have systems that gather all of this data
  – can aggregate across servers & services, compute derived values, graph data, examine historical changes, etc.
Online Profiling

• Every server supports sampling-based hierarchical profiling
  – CPU
  – memory usage
  – lock contention time

• Example: memory sampling
  – every Nth byte allocated, record stack trace of where allocation occurred
  – when sampled allocation is freed, drop stack trace
  – (N is large enough that overhead is small)
Memory Profile

- Storage D
  - StickyExecutor
  - ReceiveAndLoop
    - Run
      - 0.0 (0.0%)
      - of 4.0 (0.1%)

- RPC 2
  - NetClientChannel
    - HandleRead
      - 0.5 (1.2%)
      - of 2.3 (5.2%)
    - NewConnectionClosure
      - Run
        - 0.0 (0.0%)
        - of 3.5 (0.0%)

- Server
  - InternalNewConnection
    - 0.5 (1.1%)
    - of 3.5 (8.0%)

- Storage D
  - DServers
    - ReadOp
      - GetBuffers
        - 0.0 (0.0%)
        - of 4.0 (0.2%)

- FixedBufferCacheImpl
  - GetBuffersForRead
    - 1.5 (3.4%)
    - of 4.0 (9.2%)
Request Tracing

• Every client and server gathers sample of requests
  – different sampling buckets, based on request latency

2012/09/09-11:39:21.029630       0.018978 Read (trace_id: c6143c073204f13f ...)
11:39:21.029611      -0.000019 ... RPC: 07eb70184bfff86f ... deadline:0.8526s
11:39:21.029611      -0.000019 ... header:<path:"..." length:33082 offset:
3037807
11:39:21.029729        .    99 ... StartRead(..., 3037807, 33082)
11:39:21.029730        .     1 ... ContentLock
11:39:21.029732        .     2 ... GotContentLock
...,
11:39:21.029916        .     2 ... IssueRead
11:39:21.048196        . 18280 ... HandleRead: OK
11:39:21.048666        .   431 ... RPC: OK [33082 bytes]

• Dapper: cross-machine view of preceding information
  – can understand complex behavior across many services
  – [Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Sigelman et al., 2010]
Google’s Computational Environment Today

- Many datacenters around the world
Lots of machines...
Save a bit of power: turn out the lights...
Cool...
Cluster-Level Services

• Our earliest systems made things easier within a cluster:
  – **GFS/Colossus**: reliable cluster-level file system
  – **MapReduce**: reliable large-scale computations
  – **Cluster scheduling system**: abstracted individual machines
  – **BigTable**: automatic scaling of higher-level structured storage

• Solve many problems, but leave many cross-cluster issues to human-level operators
  – different copies of same dataset have different names
  – moving or deploying new service replicas is labor intensive
Spanner: Worldwide Storage
Spanner: Worldwide Storage

- Single global namespace for data
- Consistent replication across datacenters using Paxos
  - configurable replica placement
- Automatic migration to meet various constraints
  - resource constraints
    "The file system in this Belgian datacenter is getting full..."
  - application-level hints
    "Place 2 copies of this data in Europe and 3 copies in the U.S."
    "Place this data in flash, and place this other data on disk"

- System underlies Google’s production advertising system, among other uses

[Spanner: Google’s Globally-Distributed Database, Corbett, Dean, et al., OSDI 2012]
Shared Environment

- Random app
- CPU intensive job
- Random MapReduce #1
- Various other system services
- File system chunkserver
- Scheduling system
- Linux
- Bigtable tablet server
Shared Environment

• **Huge benefit**: greatly increased utilization

• **... but hard to predict effects increase variability**
  – network congestion
  – background activities
  – bursts of foreground activity
  – not just your jobs, but everyone else’s jobs, too
  – not static: change happening constantly

• Exacerbated by large fanout systems
• Server with 10 ms avg. but 1 sec 99%ile latency
  – touch 1 of these: 1% of requests take ≥1 sec
  – touch 100 of these: 63% of requests take ≥1 sec
Basic Latency Reduction Techniques

• Differentiated service classes
  – prioritized request queues in servers
  – prioritized network traffic

• Reduce head-of-line blocking
  – break large requests into sequence of small requests

• Manage expensive background activities
  – e.g. log compaction in distributed storage systems
  – rate limit activity
  – defer expensive activity until load is lower
Pattern: Synchronized Disruption

- Large systems often have background daemons—various monitoring and system maintenance tasks

- Initial intuition: randomize when each machine performs these tasks—actually a very bad idea for high fanout services
  - at any given moment, at least one or a few machines are slow

- Better to actually synchronize the disruptions—run every five minutes “on the dot”
  - one synchronized blip better than unsynchronized
Tolerating Faults vs. Tolerating Variability

• Tolerating faults:
  – rely on extra resources
    • RAIDed disks, ECC memory, dist. system components, etc.
  – make a reliable whole out of unreliable parts

• Tolerating variability:
  – use these same extra resources
  – make a predictable whole out of unpredictable parts

• Times scales are very different:
  – variability: 1000s of disruptions/sec, scale of milliseconds
  – faults: 10s of failures per day, scale of tens of seconds
Latency Tolerating Techniques

• **Cross request adaptation**
  – examine recent behavior
  – take action to improve latency of future requests
  – typically relate to balancing load across set of servers
  – time scale: 10s of seconds to minutes

• **Within request adaptation**
  – cope with slow subsystems in context of higher level request
  – time scale: right now, while user is waiting
Handling Within-Request Variability

• Take action within single high-level request

• Goals:
  – reduce overall latency
  – don’t increase resource use too much
  – keep serving systems safe
Pattern: Backup Requests

Replica 1:
- req 3
- req 6
- req 9

Replica 2:
- req 5
- req 9
- reply

Replica 3:
- req 8

Request 9 is canceled by replica 2.
In-memory BigTable lookups
- data replicated in two in-memory tables
- issue requests for 1000 keys spread across 100 tablets
- measure elapsed time until data for last key arrives

<table>
<thead>
<tr>
<th></th>
<th>Avg</th>
<th>Std Dev</th>
<th>95%ile</th>
<th>99%ile</th>
<th>99.9%ile</th>
</tr>
</thead>
<tbody>
<tr>
<td>No backups</td>
<td>33 ms</td>
<td>1524 ms</td>
<td>24 ms</td>
<td>52 ms</td>
<td>994 ms</td>
</tr>
<tr>
<td>Backup after 10 ms</td>
<td>14 ms</td>
<td>4 ms</td>
<td>20 ms</td>
<td>23 ms</td>
<td>50 ms</td>
</tr>
<tr>
<td>Backup after 50 ms</td>
<td>16 ms</td>
<td>12 ms</td>
<td>57 ms</td>
<td>63 ms</td>
<td>68 ms</td>
</tr>
</tbody>
</table>

- Modest increase in request load:
  - 10 ms delay: <5% extra requests; 50 ms delay: <1%
Backup Requests w/ Cross-Server Cancellation

Each request identifies other server(s) to which request might be sent.
Backup Requests: Bad Case

Server 1

- req 3
- req 9
  - also: server 2

Server 2

- req 5
- req 9
  - also: server 1

“Server 1: Starting req 9”

“Server 2: Starting req 9”

“Server 1: reply”

“Server 2: reply”
Backup Requests w/ Cross-Server Cancellation

- Read operations in distributed file system client
  - send request to first replica
  - wait 2 ms, and send to second replica
  - servers cancel request on other replica when starting read
- Time for bigtable monitoring ops that touch disk

<table>
<thead>
<tr>
<th>Cluster state</th>
<th>Policy</th>
<th>50%ile</th>
<th>90%ile</th>
<th>99%ile</th>
<th>99.9%ile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostly idle</td>
<td>No backups</td>
<td>19 ms</td>
<td>38 ms</td>
<td>67 ms</td>
<td>98 ms</td>
</tr>
<tr>
<td></td>
<td>Backup after 2 ms</td>
<td>16 ms</td>
<td>28 ms</td>
<td>38 ms</td>
<td>51 ms</td>
</tr>
<tr>
<td>+Terasort</td>
<td>No backups</td>
<td>24 ms</td>
<td>56 ms</td>
<td>108 ms</td>
<td>159 ms</td>
</tr>
<tr>
<td></td>
<td>Backup after 2 ms</td>
<td>19 ms</td>
<td>35 ms</td>
<td>67 ms</td>
<td>108 ms</td>
</tr>
</tbody>
</table>

Backups w/big sort job vs. backups w/ idle cluster!

Backups cause about ~1% extra disk reads.

-43%"
Backup Request Variants

• Many variants possible:

• Send to third replica after longer delay
  – sending to two gives almost all the benefit, however.

• Keep requests in other queues, but reduce priority

• Can handle Reed-Solomon reconstruction similarly
Pattern: Tainted Partial Results

• Many systems can tolerate inexact results
  – information retrieval systems
    • search 99.9% of docs in 200ms better than 100% in 1000ms
  – complex web pages with many sub-components
    • e.g. okay to skip spelling correction service if it is slow

• Design to proactively abandon slow subsystems
  – set cutoffs dynamically based on recent measurements
    • can tradeoff completeness vs. responsiveness
  – important to mark such results as tainted in caches
Further reading (these + more at http://research.google.com/papers)