Google™ 11 IO

# Scaling App Engine Applications

Justin Haugh, Guido van Rossum
May 10, 2011

## First things first

Justin Haugh
Software Engineer
Systems Infrastructure
jhaugh@google.com

Guido Van Rossum
Software Engineer
Tools & Runtimes
guido@google.com

Session: http://goo.gl/io/PWIEy
Hashtags: #io2011 #AppEngine
Feedback: http://www.speakermeter.com/talks/scaling-app-engine/

## Agenda

I. Scaling on App Engine
- What is scaling?
- Is scaling hard?
- The App Engine Platform
- The Scaling Formula

II. Building a Scalable App
- Scaling Tools
- Scaling Advice
- Scaling Pitfalls
- Lessons

III. Q & A

## Why do we care?

"The goal is to make it easy to get started with a new web app, and then make it easy to scale when that app reaches the point where it's receiving significant traffic and has millions of users."

App Engine Blog, April 7th, 2008

## What is scaling?

Scaling means...

- high QPS
- low latency
- no errors
- happy users, engineers, investors, etc.

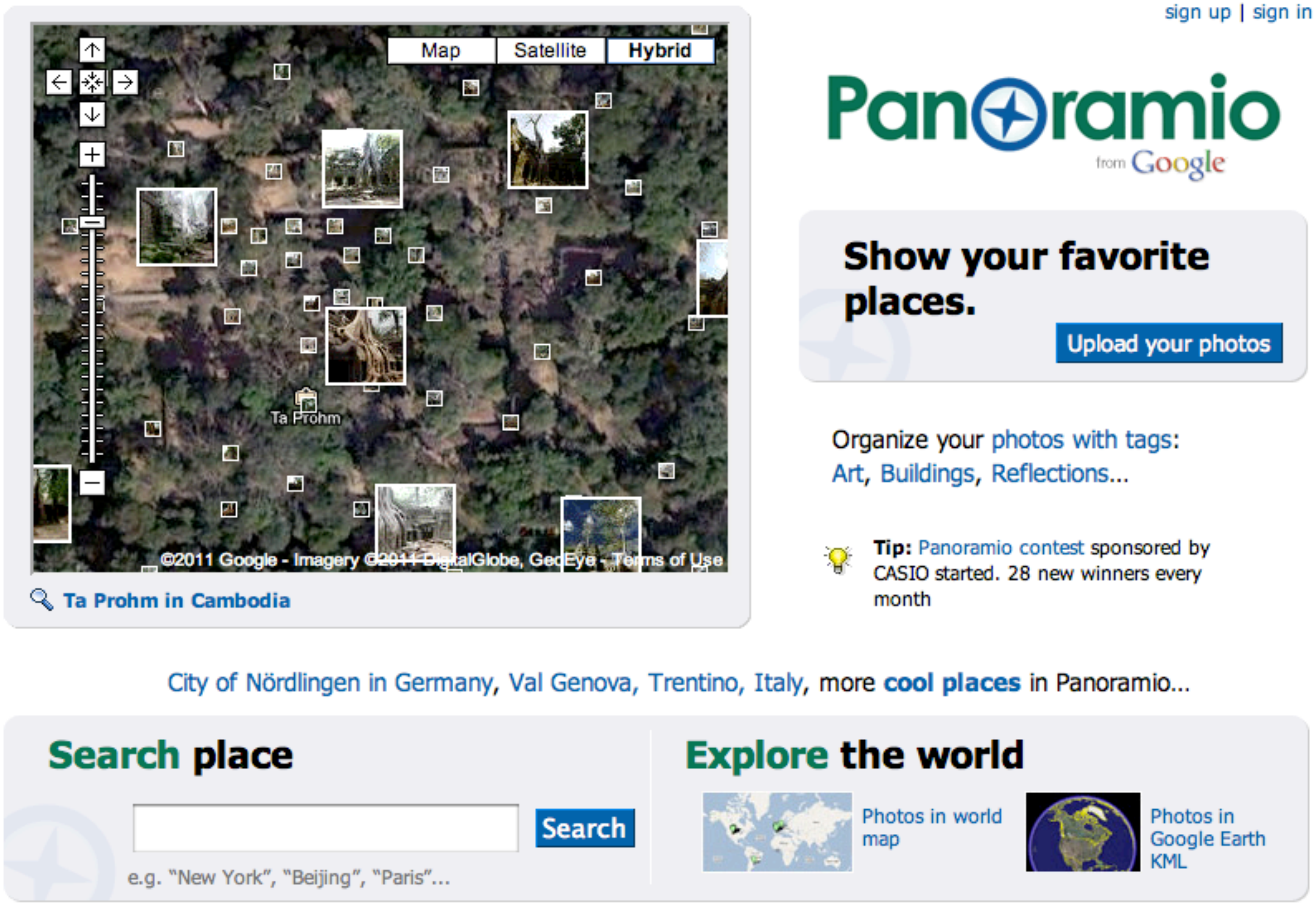
Terms

- QPS - queries per second
- latency - time to send a response
- errors - HTTP Status 500+

Numbers

- 1 QPS: easy
- 100 QPS: moderate
- 10000 QPS: impressive

## What is scaling?



sign up | sign in

Panoramio from Google

Map
Satellite
Hybrid
Ta Prohm
©2011 Google - Imagery ©2011 DigitalGlobe, GeoEye - Terms of Use

Ta Prohm in Cambodia

**Show your favorite places.**

Upload your photos

Organize your photos with tags: Art, Buildings, Reflections...

**Tip:** Panoramio contest sponsored by CASIO started. 28 new winners every month

City of Nördlingen in Germany, Val Genova, Trentino, Italy, more **cool places** in Panoramio...

**Search place**

Search

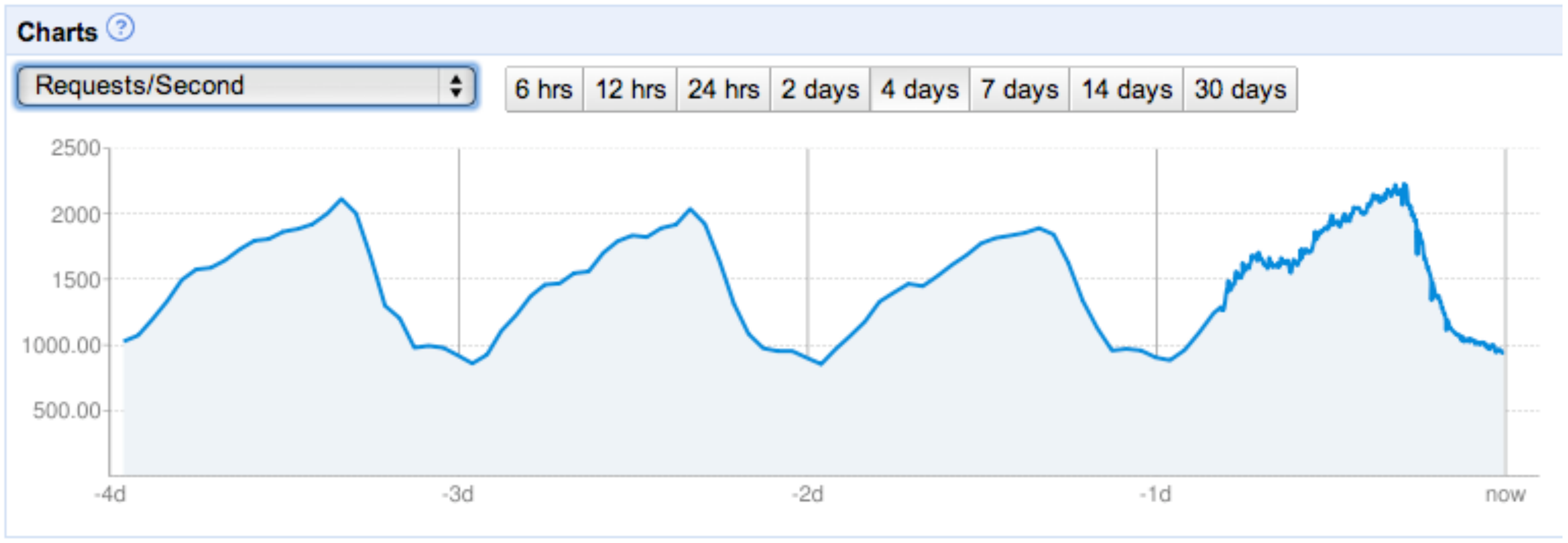
e.g. "New York", "Beijing", "Paris"...

**Explore the world**

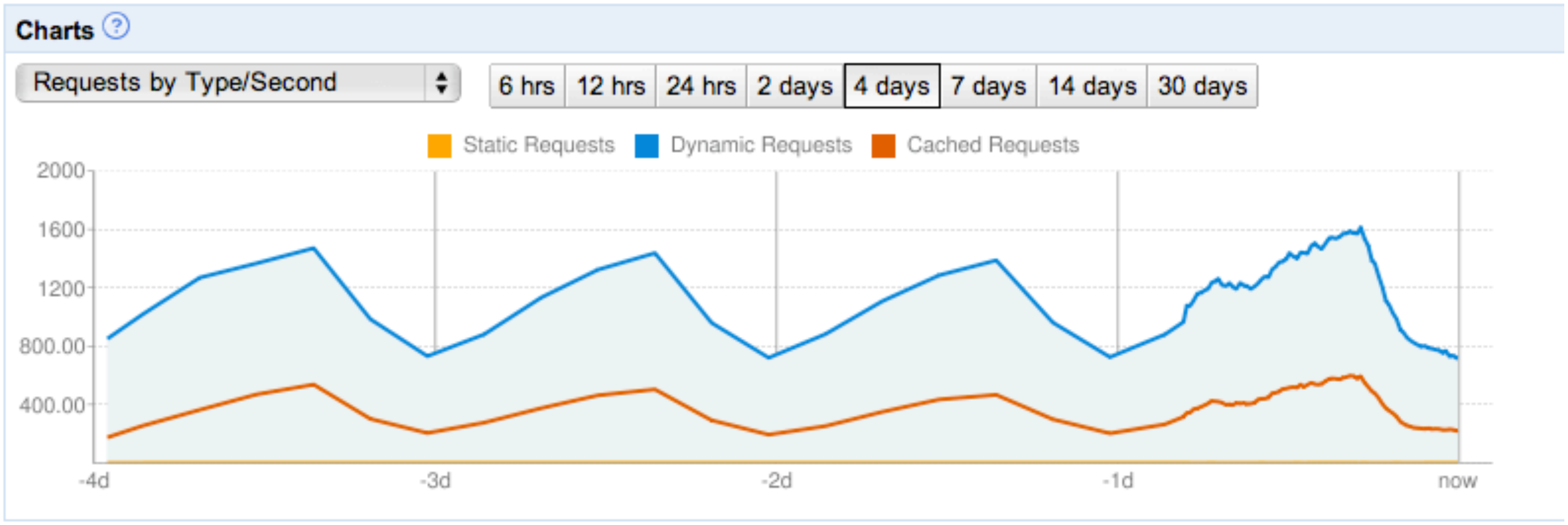
Photos in world map

Photos in Google Earth KML

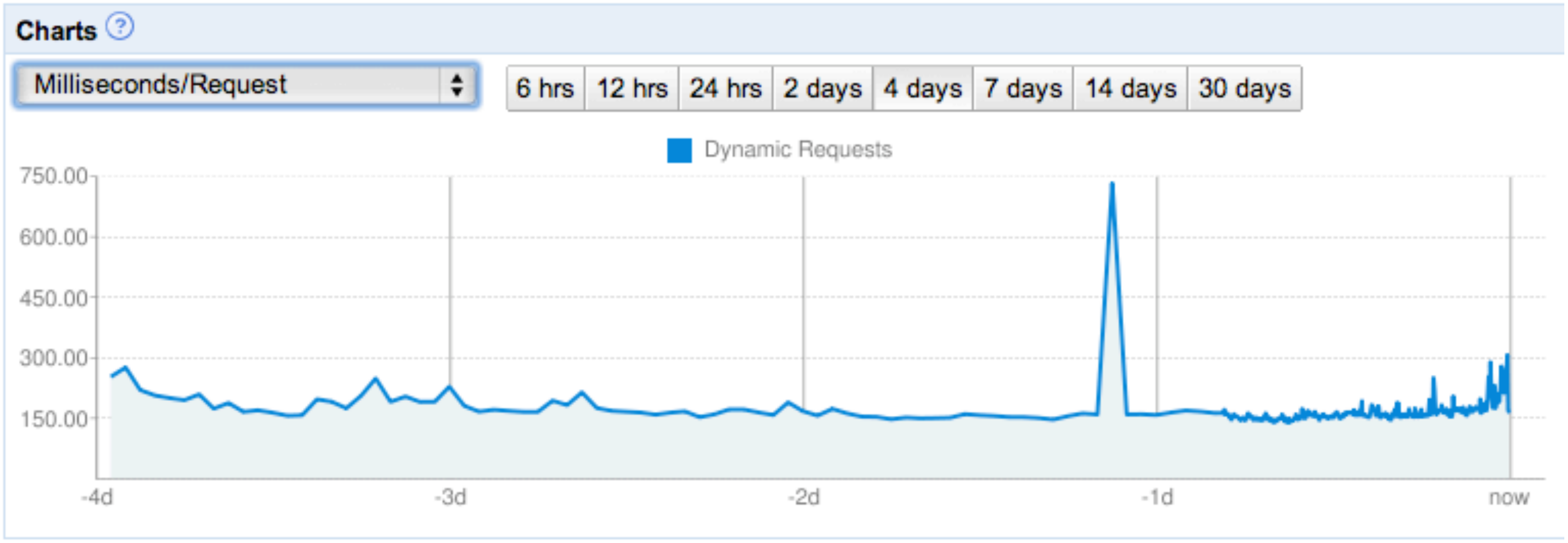
## What is scaling?



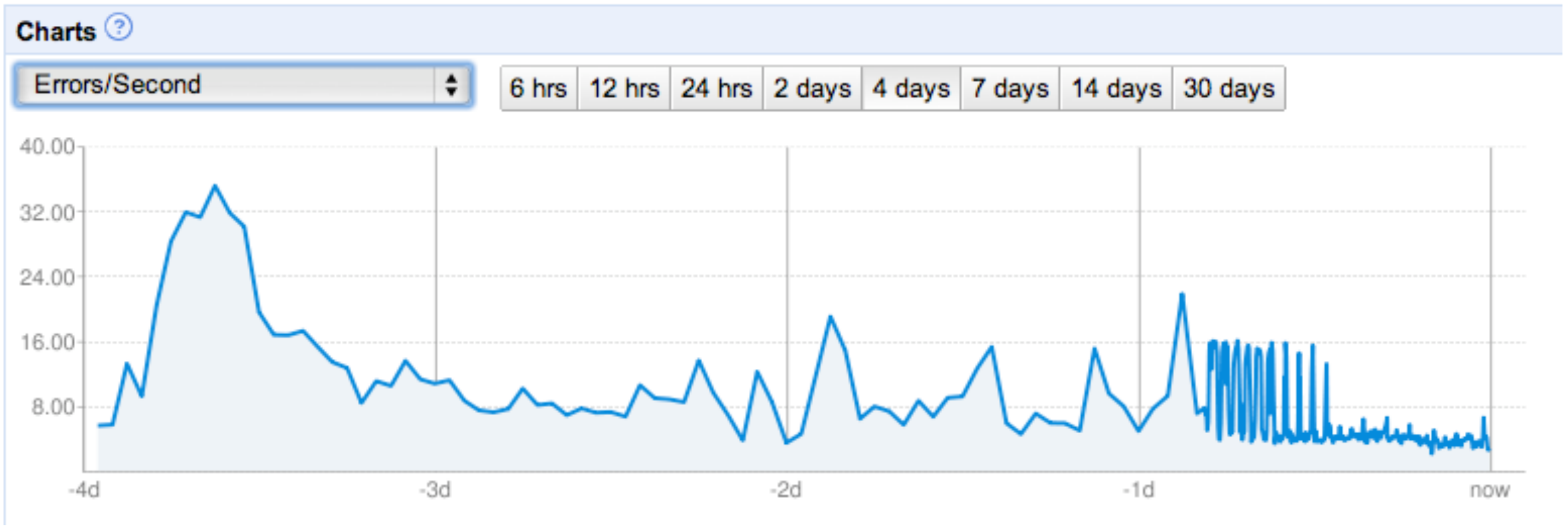
## What is scaling?



## What is scaling?



## What is scaling?



## Is scaling hard?

Two schools of thought

- No
  - most programs scale easily
  - mostly images, static files
  - read/write simple form fields
- Yes
  - anything interesting is hard
  - aggregation, data joins, fan-in, fan-out
  - complex data structures
  - audio, video, images

## Is scaling hard?

The Truth: it depends.

- on the problem you're solving
    - serve HTML, images
    - online game server
    - web search
- on the infrastructure you're using
    - your laptop
    - rack of machines
    - scalable cloud
- how much time & money you have
    - throw engineers at it
    - throw servers at it

## Is scaling hard?

Programs are born highly scalable!

- print 'hello world'
- adding things makes them less scalable

Things that slow you down

- reading or writing data
- making HTTP requests
- large requests, responses (> 10KB)
- images, audio, video

Can you do it once? (caching)
Can you do it in parallel? (async APIs)
Can you do it offline? (create a task)

## The App Engine Platform

## The App Engine Platform

- HTTP requests
  - entirely web-based
- App instances
  - python (single-threaded)
    - serial request handling
  - java (single or multi-threaded)
    - parallel request handling
- Dynamic scaling
  - instances on demand
  - if (scaling formula): add instance
- Platform goals
  - minimize pending latency
  - minimize instances
  - maximize utilization

## The Scaling Formula

- App Engine has to make a decision
  - wait for instance
  - new instance
- Inputs
  - number of instances
  - throughput of instances
  - number of requests waiting
- Predict
  - how long will it take to serve?
- Compare to load time

```
if (load_time < wait_time) {
  instance = new Instance(app);
  instances[app].insert(instance);
  instance.handle_request(request);
}
```

## The Scaling Formula

Sample Application

- latency: 100ms
- load time: 1s
- 5 instances

Case #1

- 10 requests queued
- wait time: ~200ms
- result: wait for instance

Case #2

- 100 requests queued
- wait time: ~2s
- result: load new instance

## The Scaling Formula

Load vs. Wait

- not enough
- also need: warm latency
- if wait time > warm latency:
  - new instance
- in steady-state:
  - X% of instances are idle
  - wait time << warm latency

Warmup Requests

- spin up new instance in background
- /_ah/warmup
- users never see this!
- enable in app.yaml

```
inbound_services:
- warmup
```

## The Scaling Formula

How quickly can you scale?

- depends on latency
  - 100ms: excellent
  - 250ms: good
  - 500ms: okay
  - >1s: bad
- depends on loading time
  - < 1s: good
  - 1-5s: moderate
  - > 5s: slow
- at 200ms warm latency
  - 0-10 QPS in ~1s
  - 0-100 QPS in ~10 seconds
  - 0-1000 QPS in ~5 min

## The Scaling Formula

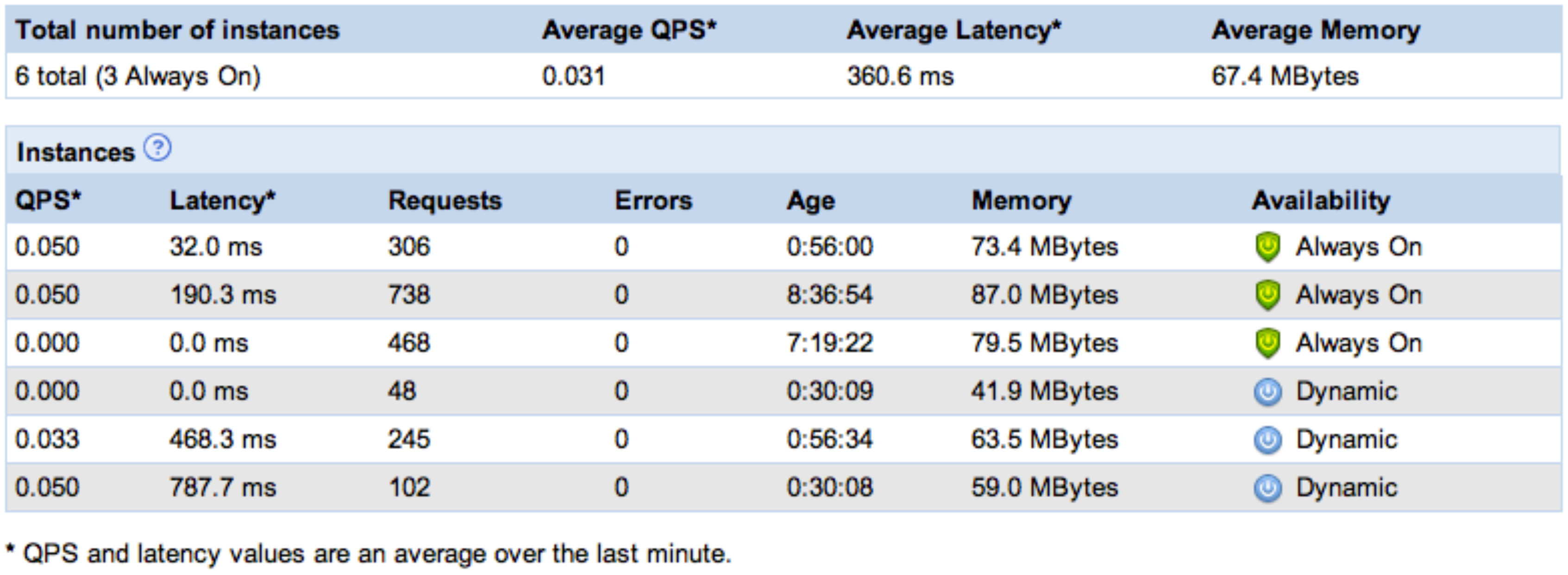
Single-Threaded (python or java)

- 1 concurrent request
- QPS vs. Latency
  - 10 ms = 100 QPS / instance
  - 100ms = 10 QPS / instance
  - 1000ms = 1 QPS / instance

Multi-Threaded (java only)

- N concurrent requests
- QPS vs. CPU (2.4 GHz)
  - 10 Mcycles / request = 240 QPS / instance
  - 100 Mcycles / request = 24 QPS / instance
  - 1000 Mcycles / request = 2.4 QPS / instance

## Instances Console



| Total number of instances | Average QPS* | Average Latency* | Average Memory |
|---|---|---|---|
| 6 total (3 Always On) | 0.031 | 360.6 ms | 67.4 MBytes |

**Instances**

| QPS* | Latency* | Requests | Errors | Age | Memory | Availability |
|---|---|---|---|---|---|---|
| 0.050 | 32.0 ms | 306 | 0 | 0:56:00 | 73.4 MBytes | Always On |
| 0.050 | 190.3 ms | 738 | 0 | 8:36:54 | 87.0 MBytes | Always On |
| 0.000 | 0.0 ms | 468 | 0 | 7:19:22 | 79.5 MBytes | Always On |
| 0.000 | 0.0 ms | 48 | 0 | 0:30:09 | 41.9 MBytes | Dynamic |
| 0.033 | 468.3 ms | 245 | 0 | 0:56:34 | 63.5 MBytes | Dynamic |
| 0.050 | 787.7 ms | 102 | 0 | 0:30:08 | 59.0 MBytes | Dynamic |

* QPS and latency values are an average over the last minute.

## Scaling in Reality

Back to Reality

- theoretical QPS limits not achievable
  - routing overhead
  - load fluctuations
  - safety limits
- we tweak the formula regularly
  - performance vs. utilization
  - machine upgrades
  - infrastructure changes
    - precompilation
    - warming requests

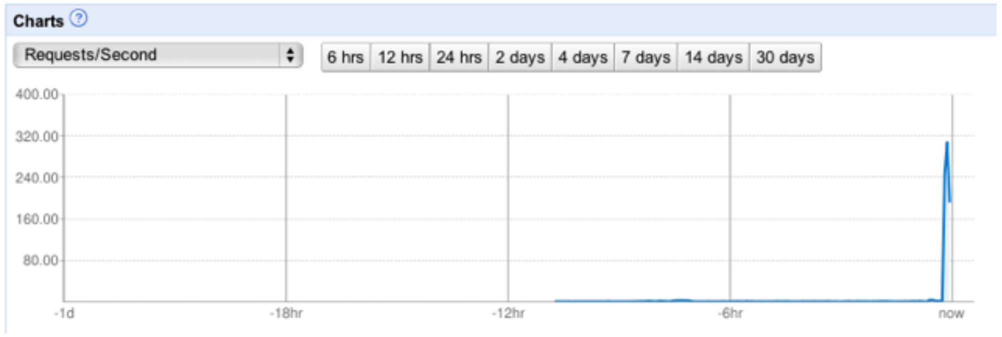
## Scaling in Reality

Takeaways

- App Engine does:
  - track latency, cpu
  - add/remove instances
  - optimize performance, utilization
  - respond quickly to traffic spikes
  - save you time and money
- App Engine doesn't:
  - understand your latency
  - understand your CPU
  - make your app performant
  - optimize your code

It's a partnership - App Engine scales if you do.

## Final Thoughts...



From: "Matt Mastracci" <mm@gri.pe>
Date: Nov 10, 2010 7:41 PM
Subject: AppEngine flawless scaling
To: "Fred Sauer" <fredsa@google.com>

Fred,

Just wanted to say thanks to the AppEngine team for a great product. We got a brief mention on "The View" this morning and that resulted in a nice bump like so:

AppEngine scaled wonderfully. We got about 900 errors on the front page while it scaled up, but compared to the overall traffic, that was nothing. Our app is still seeing a ton of traffic, but it's amazingly fast. You wouldn't even know!

Thanks to the AppEngine team for a great product,
Matt Mastracci & gri.pe team

## Agenda

I. Scaling on App Engine
- What is scaling?
- Is scaling hard?
- The App Engine Platform
- The Scaling Formula

II. Building a Scalable App
- Scaling Tools
- Scaling Advice
- Scaling Pitfalls
- Lessons

III. Q & A

## Measuring App Performance

- Techniques and tools
  - Load testing
  - Appstats

- Note: even measured results have a "half-life"
  - Users' behavior changes
  - Adding features changes the app's behavior
  - Data accumulates
  - App Engine itself changes

# Load Testing: What is a Load Test?

- Drive synthetic traffic to your site

- When live traffic hits a bump, it's too late to fix!

- Basic approach:
  - Slowly increase traffic until you hit a wall
    - (high error rate or no increase in QPS)
  - Use tools to understand the bottleneck
    - Dashboard, Instances console, Logs
    - Appstats (coming up)
  - Tentatively fix the bottleneck
  - Rinse and repeat

## Load Testing: Doing it Right

- Don't increase traffic too fast!
  - Instance creation will lag, error rate will peak
  - Ramp traffic up gradually

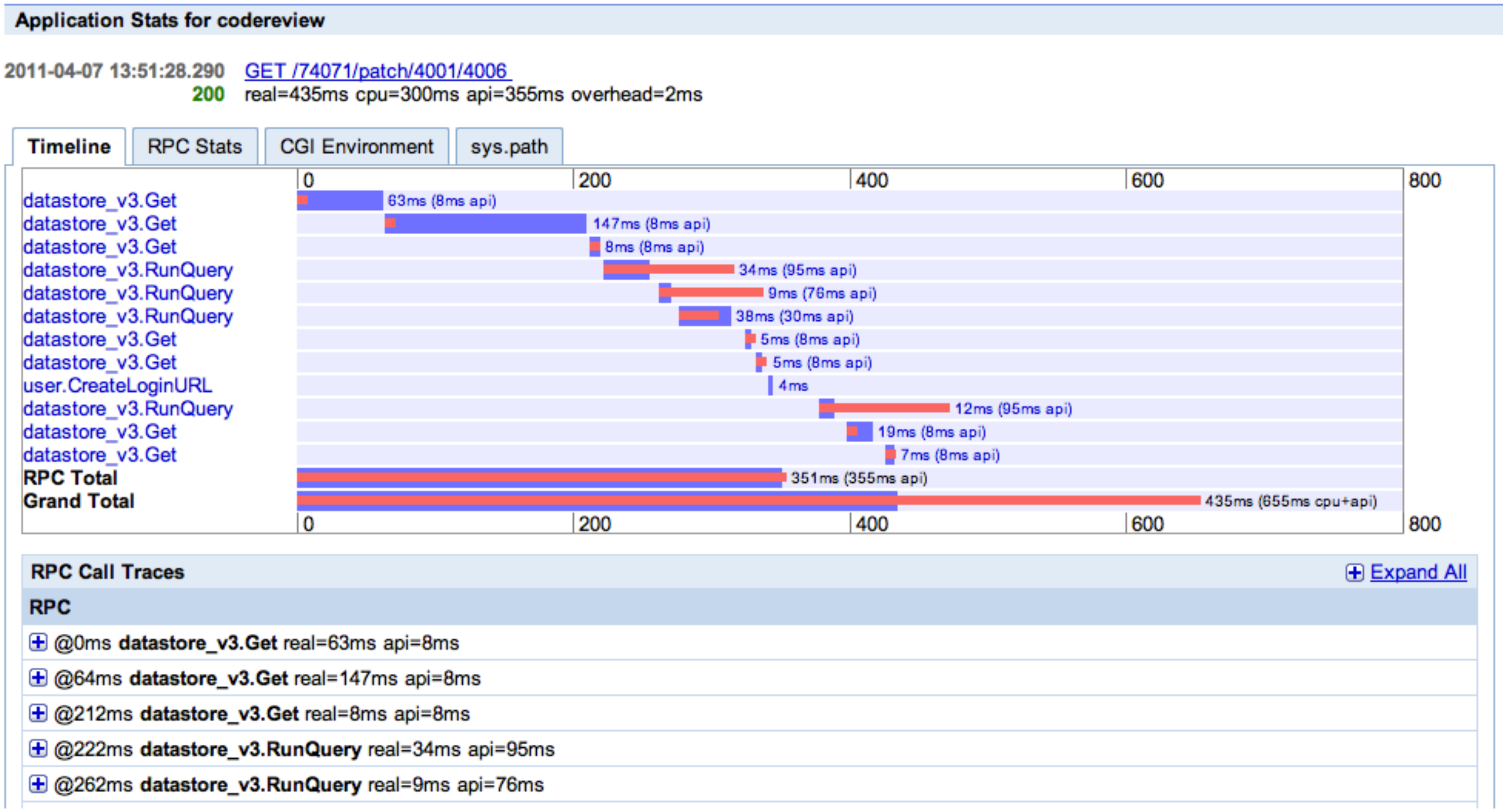
- Use realistic synthetic traffic patterns
  - E.g. record sessions from early users
  - Include requests for CSS, images etc.
    - But take client caching into account

- Beware of bottlenecks in the load generator
  - E.g. network link of load generator

## Appstats: Requests Under the Microscope!

- Usually what slows you down is too many RPC calls
  - Includes datastore, memcache, urlfetch, etc.
  - Traditional CPU profiling is relatively useless
- Appstats instruments and visualizes RPC calls
- "I was blind, now I can see"
  - (actual user comment)
- Search for: *appstats*
- Python and Java

# Appstats: UI



**Application Stats for codereview**

2011-04-07 13:51:28.290 GET /74071/patch/4001/4006
200 real=435ms cpu=300ms api=355ms overhead=2ms

| | |
|---|---|
| datastore_v3.Get | 63ms (8ms api) |
| datastore_v3.Get | 147ms (8ms api) |
| datastore_v3.Get | 8ms (8ms api) |
| datastore_v3.RunQuery | 34ms (95ms api) |
| datastore_v3.RunQuery | 9ms (76ms api) |
| datastore_v3.RunQuery | 38ms (30ms api) |
| datastore_v3.Get | 5ms (8ms api) |
| datastore_v3.Get | 5ms (8ms api) |
| user.CreateLoginURL | 4ms |
| datastore_v3.RunQuery | 12ms (95ms api) |
| datastore_v3.Get | 19ms (8ms api) |
| datastore_v3.Get | 7ms (8ms api) |
| **RPC Total** | 351ms (355ms api) |
| **Grand Total** | 435ms (655ms cpu+api) |

**RPC Call Traces** Expand All

**RPC**

@0ms **datastore_v3.Get** real=63ms api=8ms

@64ms **datastore_v3.Get** real=147ms api=8ms

@212ms **datastore_v3.Get** real=8ms api=8ms

@222ms **datastore_v3.RunQuery** real=34ms api=95ms

@262ms **datastore_v3.RunQuery** real=9ms api=76ms

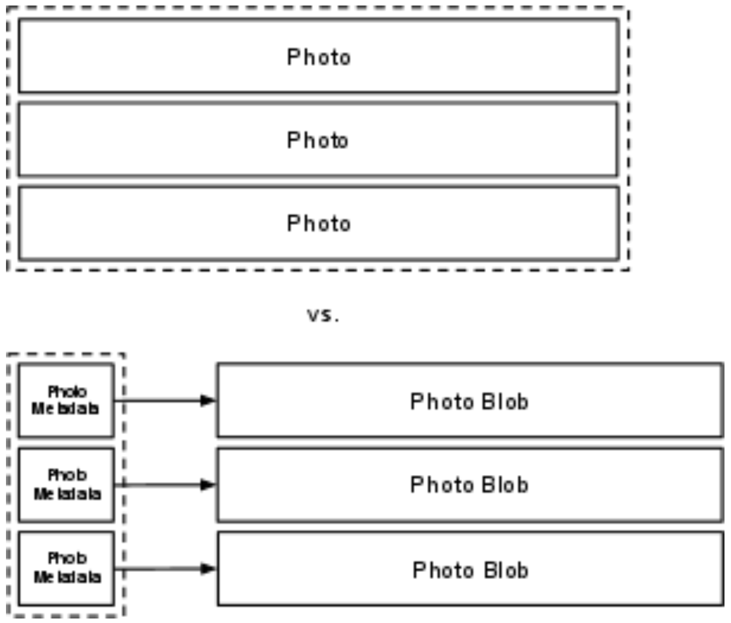
## Advice on Making Your App Scale Better

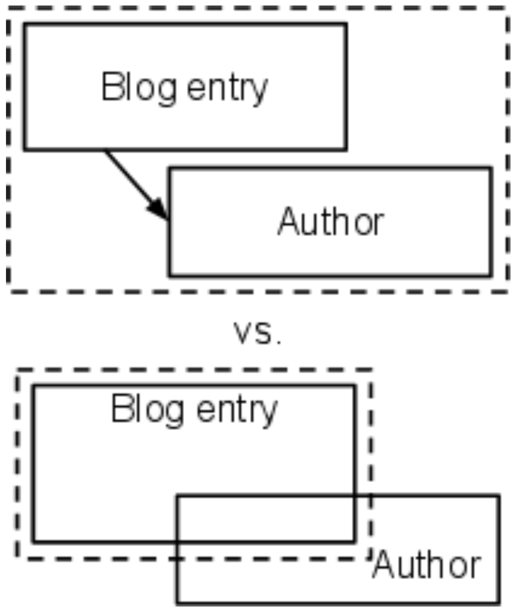
Lots of strategies available! For example:

1. Stupid schema tricks
2. Batching RPCs
3. Parallel RPCs
4. Caching, Caching, Caching!

## 1. Stupid Schema Tricks

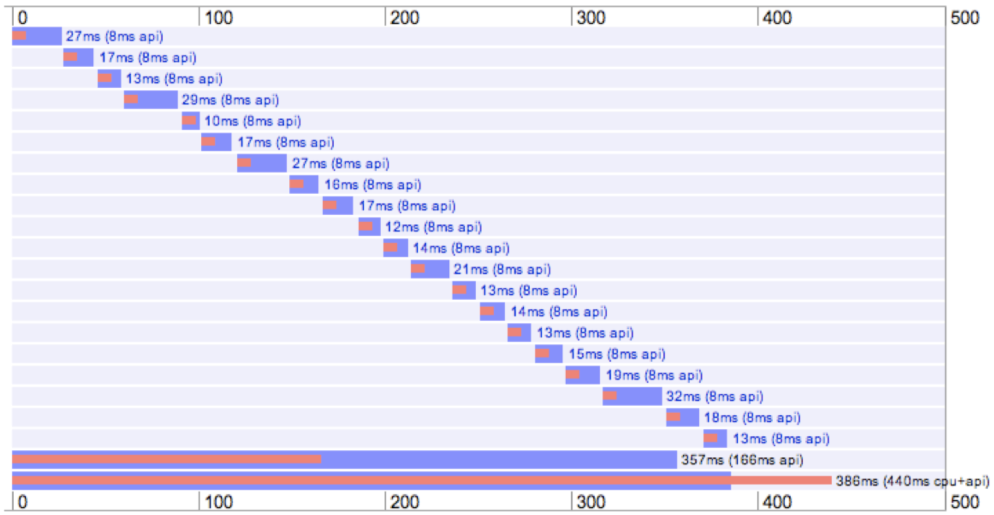
- Unlearn your SQL habits
- Examples:
  - split data across multiple entities
  - duplicate data in multiple entities

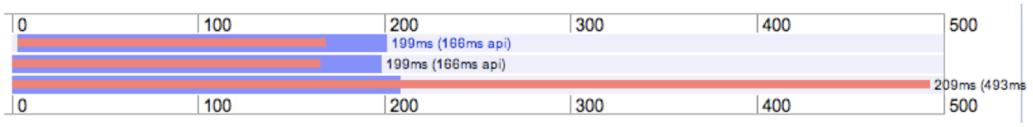




## 2. Batching

- Read or write multiple entities in one RPC
- Fewer round trips: reduced latency
  - memcache: 1-5 msec
  - datastore: 15-50 msec
- Datastore: **db.get([keys]), db.put([entities])**
- Memcache: **memcache.get_multi(...), .set_multi(...)**

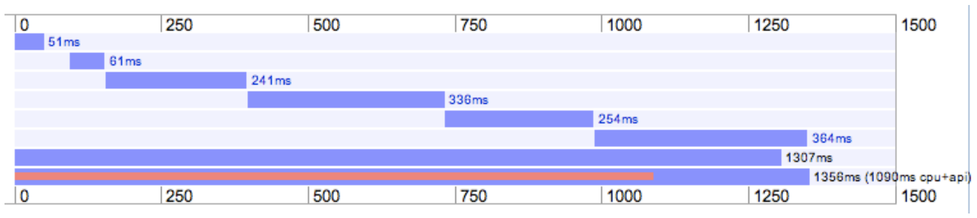


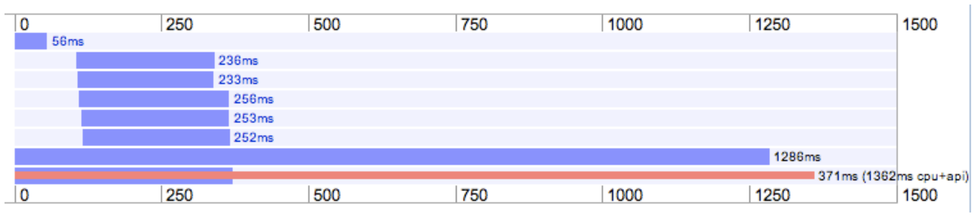


# 3. Parallel RPCs

- Parallel urlfetch: well-known
- Parallel datastore calls: new in App Engine 1.5.0
- **rpc1 = db.get_async(keys1)**
  **rpc2 = db.put_async(entities2)** # Start overlapping get, put
- **entities1 = rpc1.get_result()**
  **keys1 = rpc2.get_result()** # Wait for results as needed

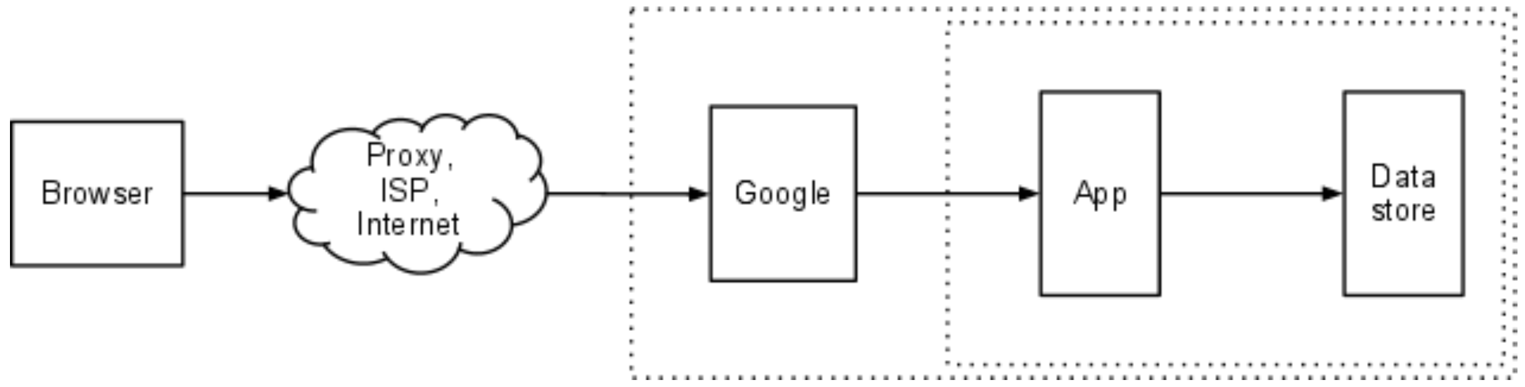
- Also available in Java, via getAsyncDatastoreService()





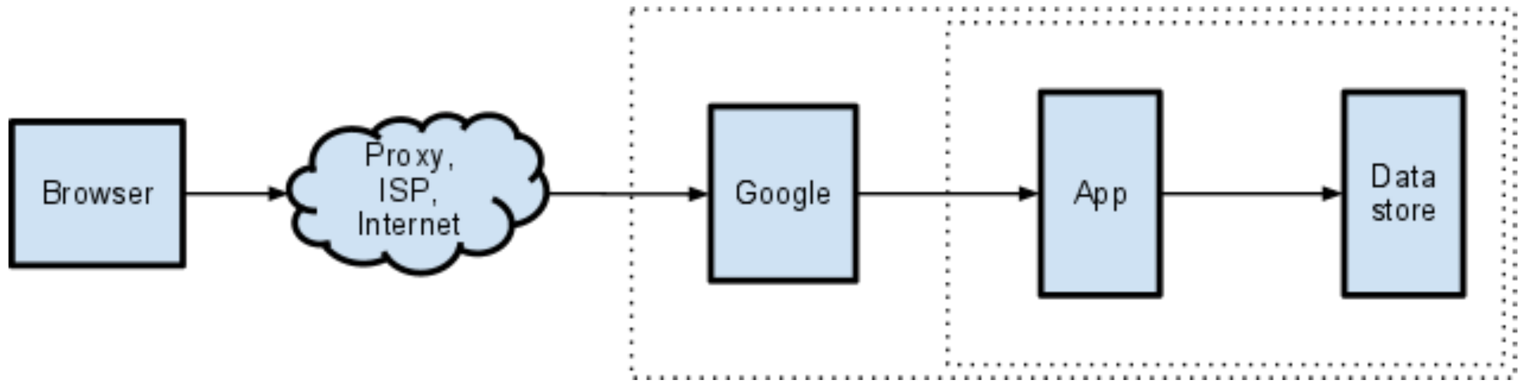
## 4. Caching, Caching, Caching!

- Where can caching happen?



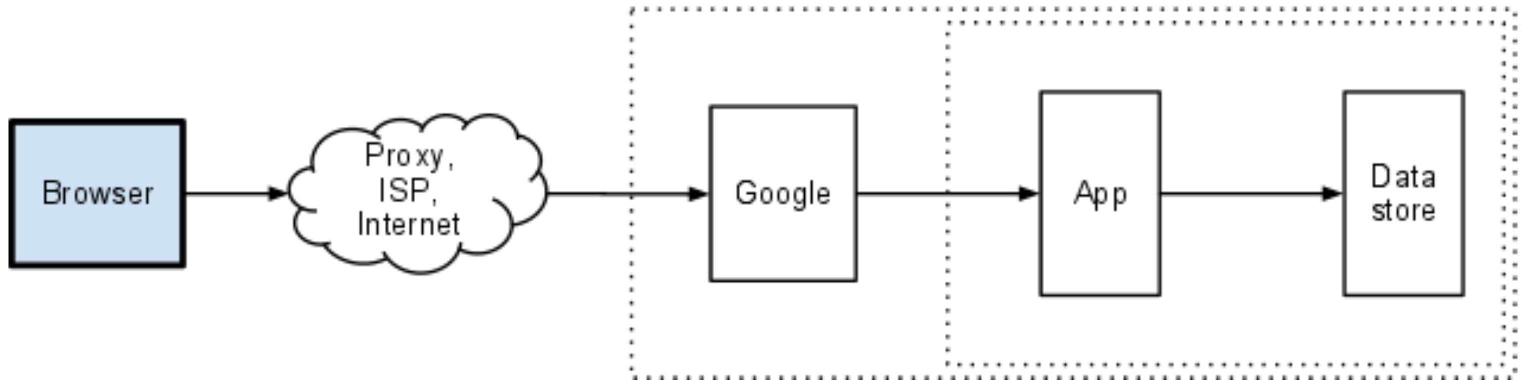
## 4. Caching, Caching, Caching!

- Where can caching happen?
- Everywhere!



## 4.1. Caching: Close to the User

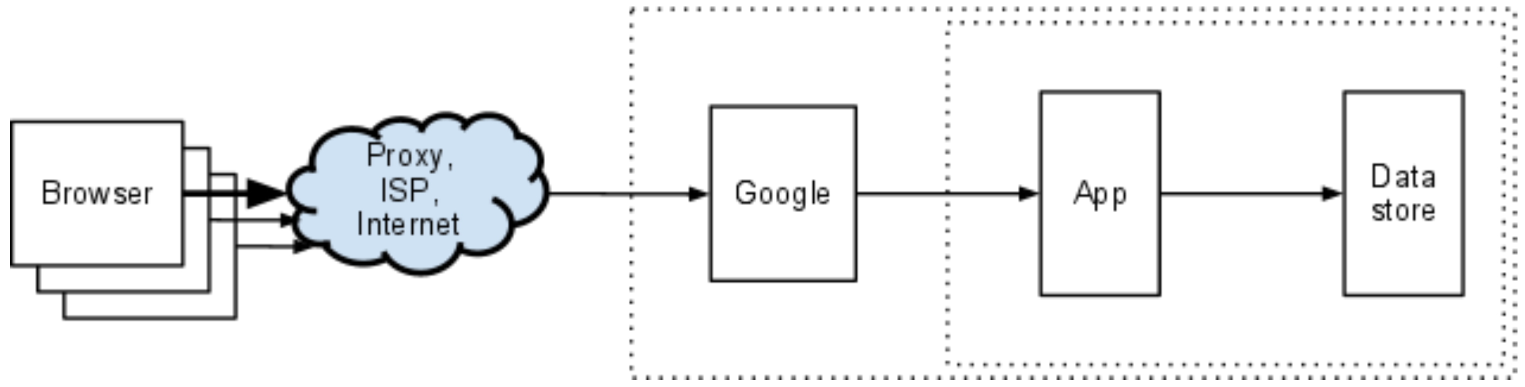
- In the browser:
- Cache-control, Etags, Expires headers



- Cache-control: **private,** max-age=60
- Helps the user more than it helps the app
- How to pick max-age???

## 4.2. Caching: In the Public Internet

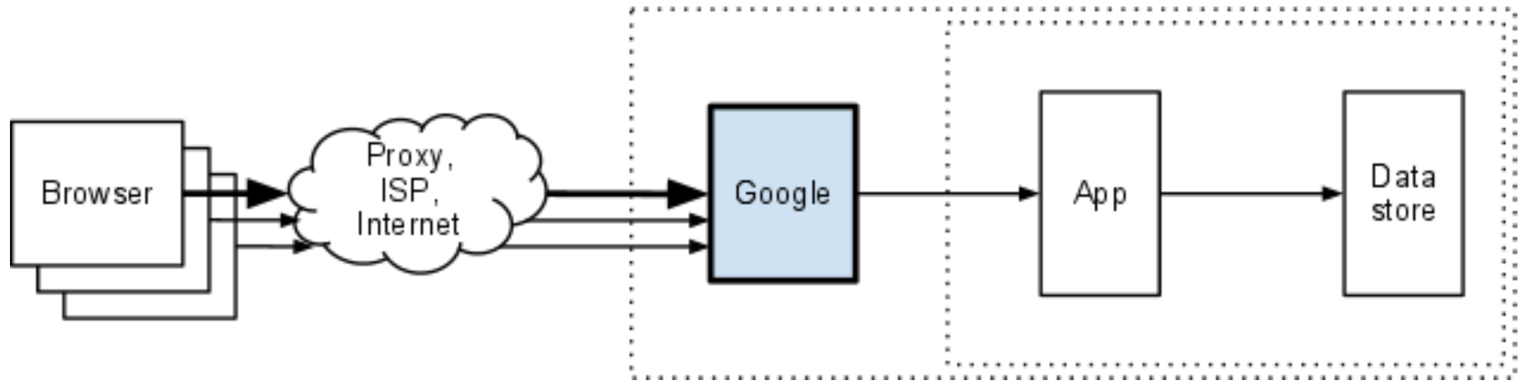
- Between browser and Google:
- Cache-control, Etags, Expires headers



- Cache-control: **public,** max-age=60
- Yes, 60 seconds is enough!
- Beware: cached data shared between users

# 4.3. Caching: At Google

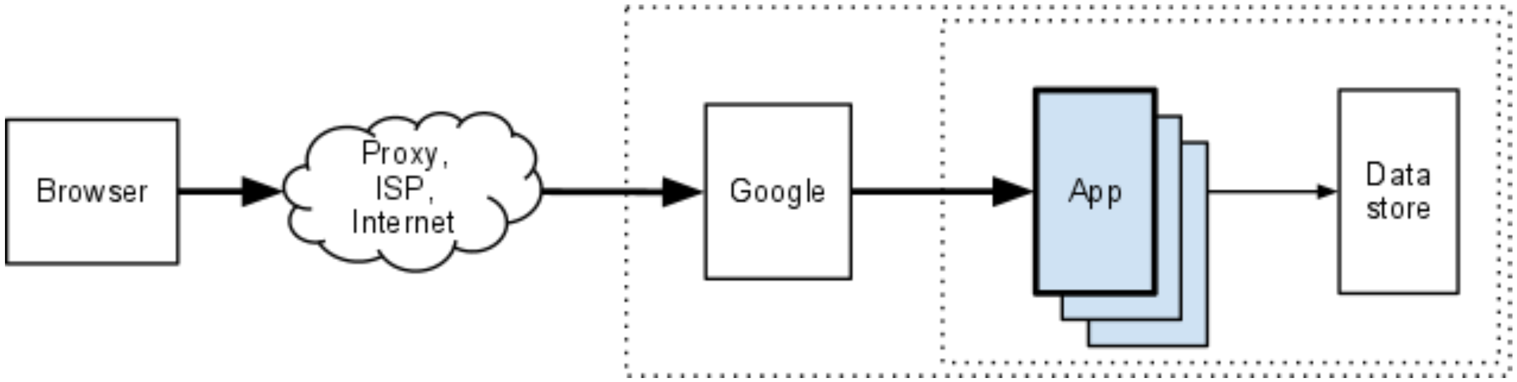
- In Google front-ends:
- Cache-control, Etags, Expires headers



- Cache-control: **public,** max-age=60
- Beware: cached data shared between users
  - No guarantees!
  - Only for paid apps
  - Dashboard: *Request by Type* chart
  - Logs: response code 204

## 4.4. Caching: In Process Memory

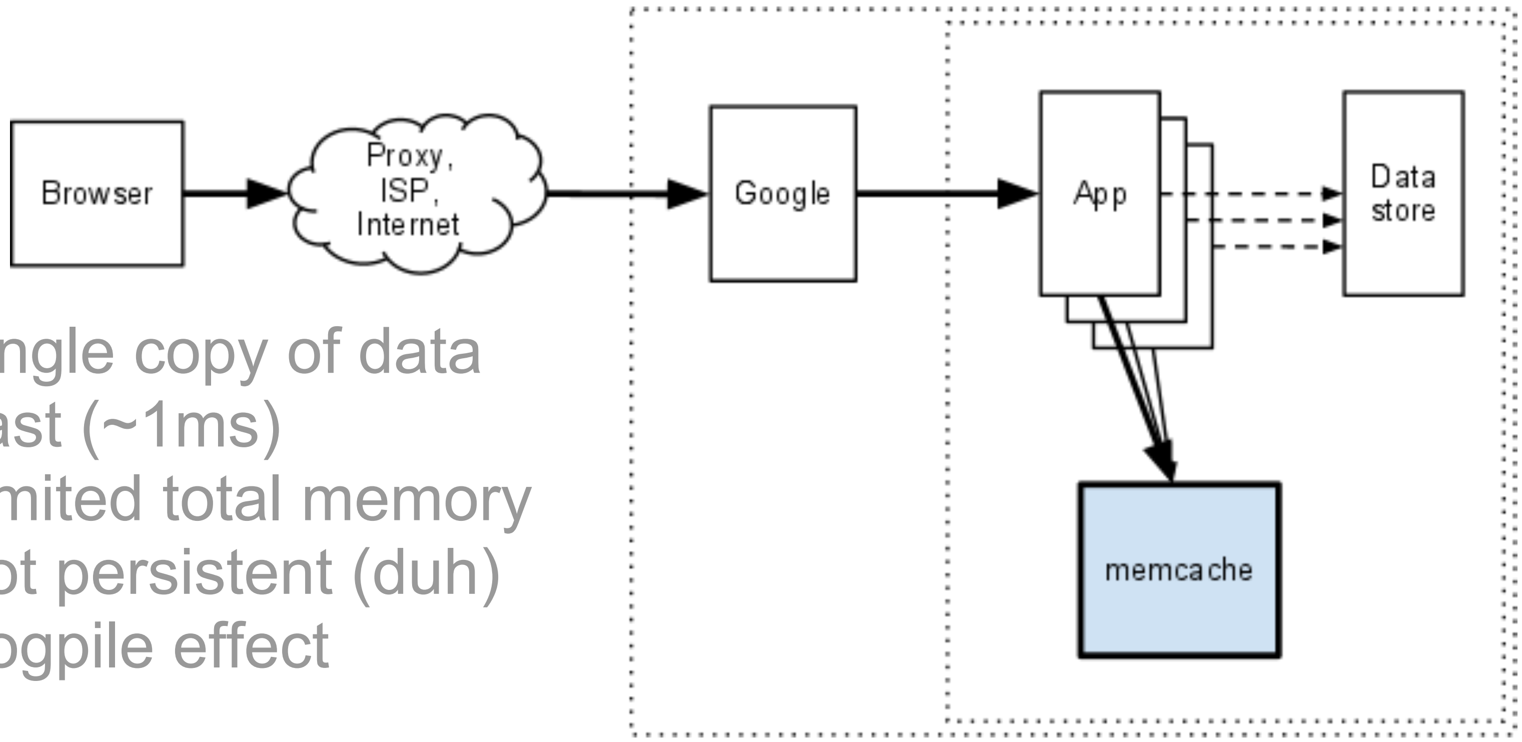
- In your app:
- In instance (process) memory



- Python: (module-) global variables
- Java: static variables (shared between threads)
- Each instance has its own copy: no consistency!

## 4.5. Caching: Memcache

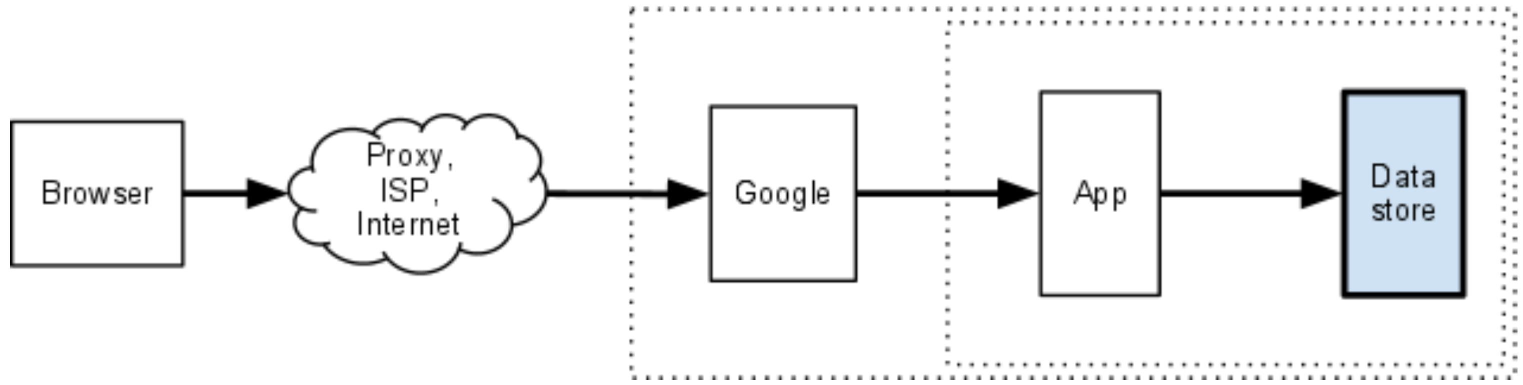
- Near your app:
- Memcache (search for: *app engine memcache*)



- Single copy of data
- Fast (~1ms)
- Limited total memory
- Not persistent (duh)
- Dogpile effect

## 4.6. Caching: In the Datastore (!)

- Near your app:
- In the datastore (yes, caching in the datastore)



- E.g. in bloggart (Nick Johnson): full page pre-generated
  - when blog entry is created or edited
  - pages served from memcache or datastore
  - off-line pre-generation with task queue

## Pitfalls

- Common performance bugs
  - See second half of last year's Appstats talk
    - Search for: *appstats i/o 2010 video*

- Calling out two specific datastore bottlenecks:
  1. Entity contention
  2. Hot tablets

## 1. Entity Contention

- Datastore limited to ~1 write/second per entity group
  - (aggregate write rate much higher of course :-)

- Most common offenders: global counters
  - Solution: *sharding counters* (search for it)
    - (#2 hit explains non-sharding alternative :-)

- Other solutions:
  - Intelligent use of *entity groups* (search for it)
  - Off-load writes to task queue
  - Use memcache if it doesn't have to be perfect
  - Use backends (see next talk in this track)

## 2. Hot Tablets: Problem Description

- Rapidly appending in sequential order
  - E.g. Kind/1000001, Kind/1000002, Kind/1000003, ...
  - Also applies to indexed property values!

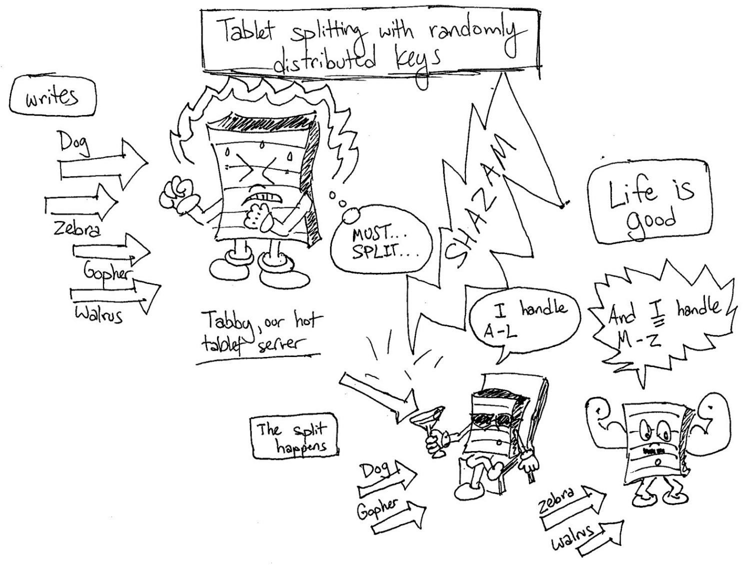
- Causes temporary worst-case Datastore behavior
  - Data is split across "tablets"
  - Sequential appends all go to same tablet
  - Tablet server limited to N writes/second

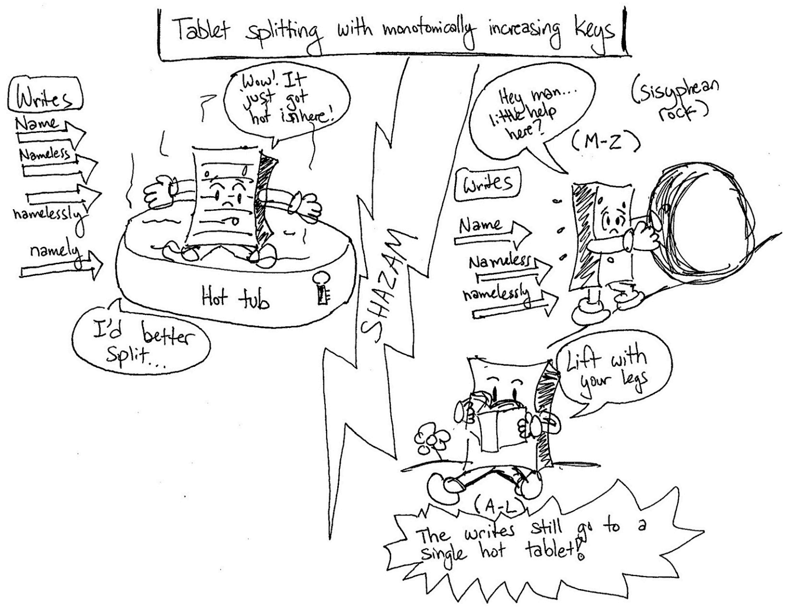
- Common symptom: datastore timeouts --> app errors

## 2. Hot Tablets: What To Do

- Solution: randomize key/data range
  - E.g. Kind/"justin", Kind/"guido", Kind/"ikai", ...
    - prepend hash or MD5 if data is not uniform

- Search for: *Ikai Lan says hot tablets*





Tablet splitting with monotonically increasing keys
Writes
Name
Nameless
namelessly
namely
Wow! It just got hot in here!
Hot tub
I'd better split...
SHAZAM
Hey man... little help here?
(Sisyphean rock)
(M-Z)
Writes
Name
Nameless
namelessly
Lift with your legs
(A-L)
The writes still go to a single hot tablet!

## Lessons

Some things we've learned:

- App Engine scales but you still must do some work :)
- New instances created as needed, gradually
- Request latency is key (less latency: more throughput)
- Treat performance tuning as science
- Tools & techniques: load testing, Appstats
- Stupid schema tricks, batching, parallel RPCs
- Caching, caching, caching!
- Common bottlenecks: entity contention and hot tablets

## Q & A (Please Use the Microphone!)

- Session: http://goo.gl/io/PWIEy
- Hashtags: #io2011 #AppEngine
- Feedback: http://www.speakermeter.com/talks/scaling-app-engine/

- Appstats talk: http://www.google.com/events/io/2010/sessions/appstatsrpc-appengine.html
- <threadsafe>: http://code.google.com/appengine/docs/java/config/appconfig.html
- Sharding counters: http://code.google.com/appengine/articles/sharding_counters.html
- Non-sharding counters: http://blog.notdot.net/2010/04/High-concurrency-counters-without-sharding
- Hot tablets: http://ikaisays.com/2011/01/25/app-engine-datastore-tip-monotonically-increasing-values-are-bad/

Google™ 11 I O