Operation (1)

November 15, 2022

Byung-Gon Chun

(Credit: Slides from UCB CS169 taught by Armando Fox, David Patterson)

Operation

- Running application services on the public cloud
  - *-as-a-service
  - IAAS, PAAS, FAAS, SAAS, DAAS, MAAS, BAAS, …
- Building up your own infrastructure (on-premise)
- Hybrid

From Development to Deployment

Outline of topics

- Continuous integration & continuous deployment
- Upgrades & feature flags
- Availability & responsiveness
- Monitoring
- Relieving pressure on the database
- Getting insight from data
- Defending customer data

Development vs. Deployment

Development:

- Testing to make sure your app works as designed

Deployment:

- Testing to make sure your app works when used in ways it was *not* designed to be used

Bad News

- “Users are a terrible thing”
- some bugs only appear under stress
- production environment != development environment
- the world is full of evil forces
- and so on

Good News

- Operating on the cloud
  - On-demand resource provisioning
  - “Easy” tiers of horizontal scaling
  - Component-level performance tuning
  - Infrastructure-level security
  - Many basic infrastructure services to use
  - …
  - E.g., Amazon Web Service, Microsoft Azure, Google Cloud

“Performance & Security” Defined

- Availability or Uptime
  What % of time is site up & accessible?
- Responsiveness
  – How long after a click does user get a response?
- Scalability
  – As # users increases, can you maintain responsiveness without increasing cost/user?

- Privacy
  – Is data access limited to the appropriate users?
- Authentication
  – Can we trust that user is who s/he claims to be?
- Data integrity
  – Is users’ sensitive data tamper-evident?

Performance
Stability

Security

Quantifying Availability and Responsiveness

Availability and Response time

- Gold standard: US public phone system, 99.999% uptime (“five nines”)
  - Rule of thumb: 5 nines ~ 5 minutes/year
  - Since each nine is an order of magnitude, 4 nines ~ 50 minutes/year, etc.
  - Good Internet services get 3-4 nines
- Response time: how long do I perceive a response after I interact with a site?
  - For small content on fast network, dominated by latency (not bandwidth)

Is response time important?

- How important is response time?*
  - Amazon: +100ms => 1% drop in sales
  - Google: +500ms => 20% fewer searches
- Classic studies (Miller 1968, Bhatti 2000)

  < 100 ms is “instantaneous”

  > 7 sec is abandonment time

Jeff Dean,
Google Fellow

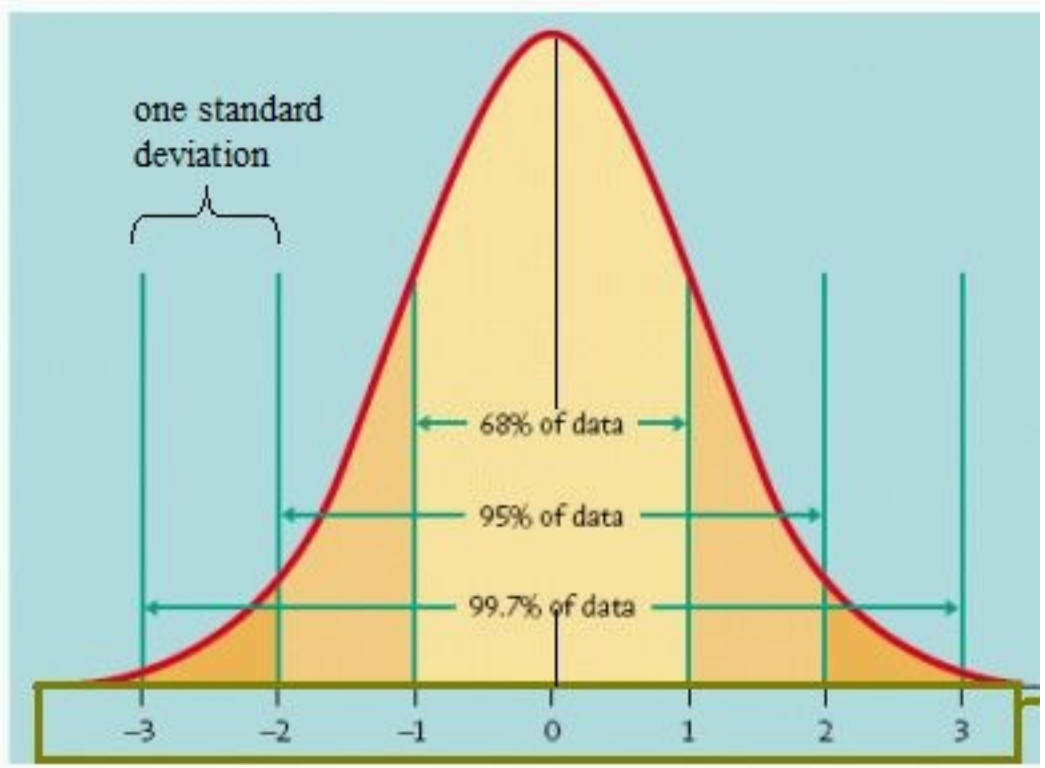
“Speed is a feature”

Source: Nicole Sullivan (Yahoo! Inc.), *Design Fast Websites*, http://www.slideshare.net/stubbornella/designing-fast-websites-presentation

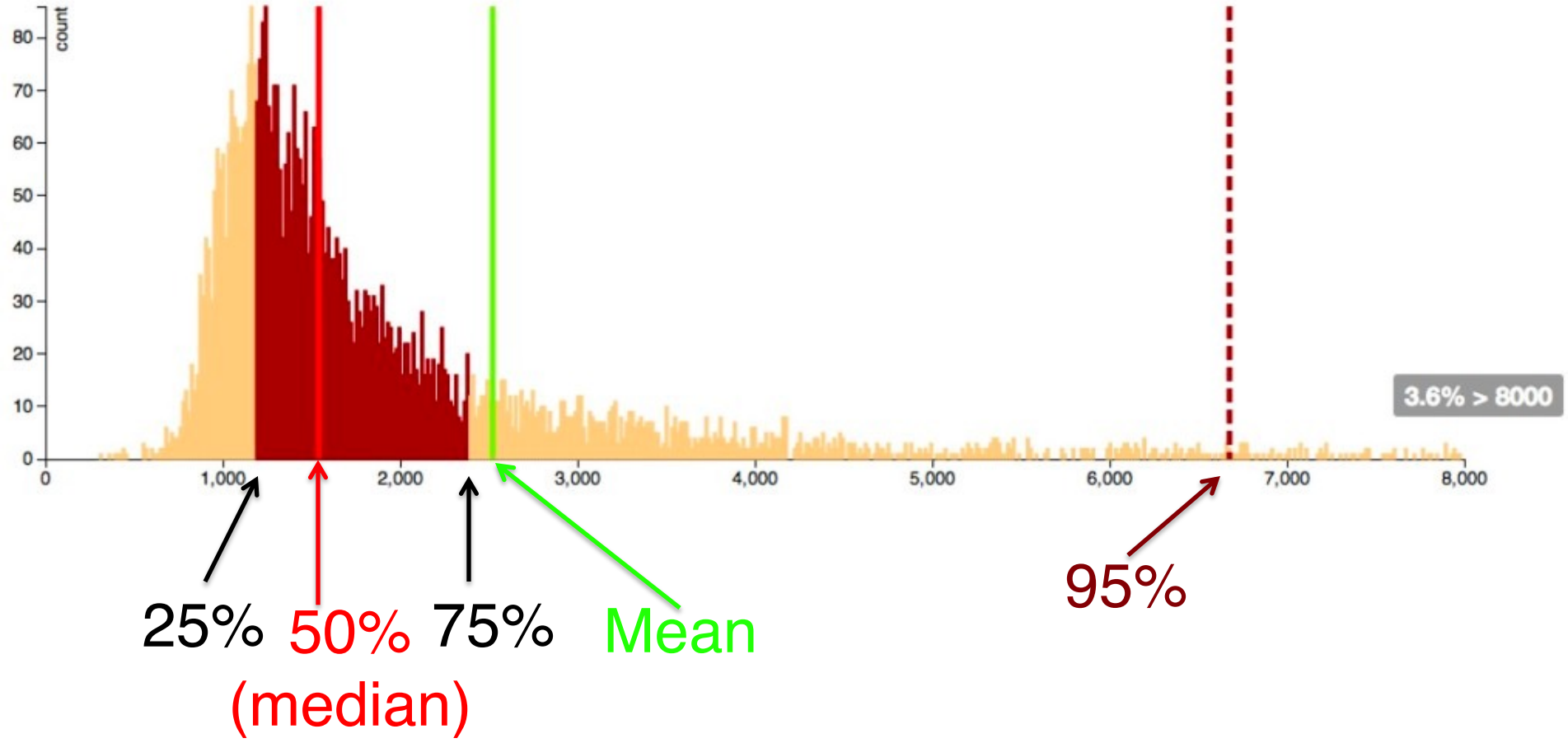
Simplified (& **false**) view of performance

- For *standard normal distribution* of response times around mean: ±2 standard deviations around mean is 95% confidence interval

- *Average* response time T means:
- 95%ile users are getting T+2σ
- 99.7% users get T+3σ



A real example



Courtesy Bill Kayser, Distinguished Engineer, New Relic. http://blog.newrelic.com/breaking-down-apdex
Used with permission of the author.

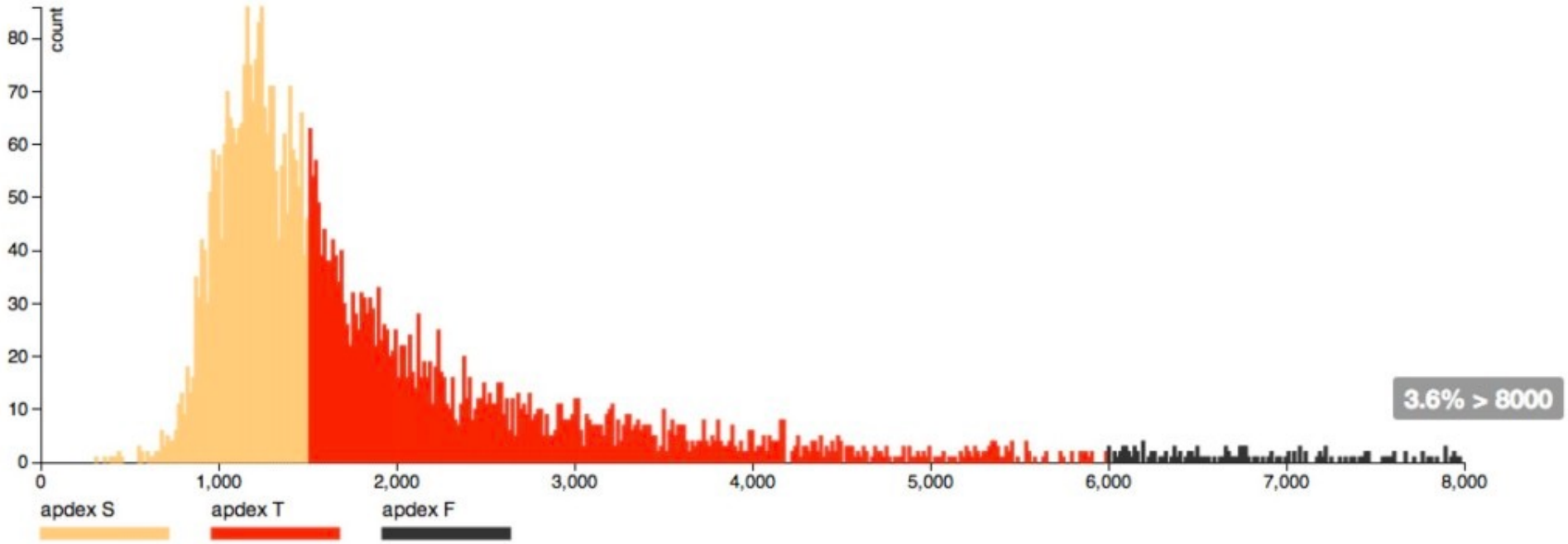
Service Level Objective (SLO)

- Time to satisfy user request
  (“latency” or “response time”)
- SLO: Instead of worst case or average: what % of users get acceptable performance
- Specify %ile, target response time, time window
  - e.g., 99% < 1 sec, over a 5 minute window
- Service level *agreement* (SLA) is an SLO to which provider is contractually obligated

Apdex (Application Performance Index)

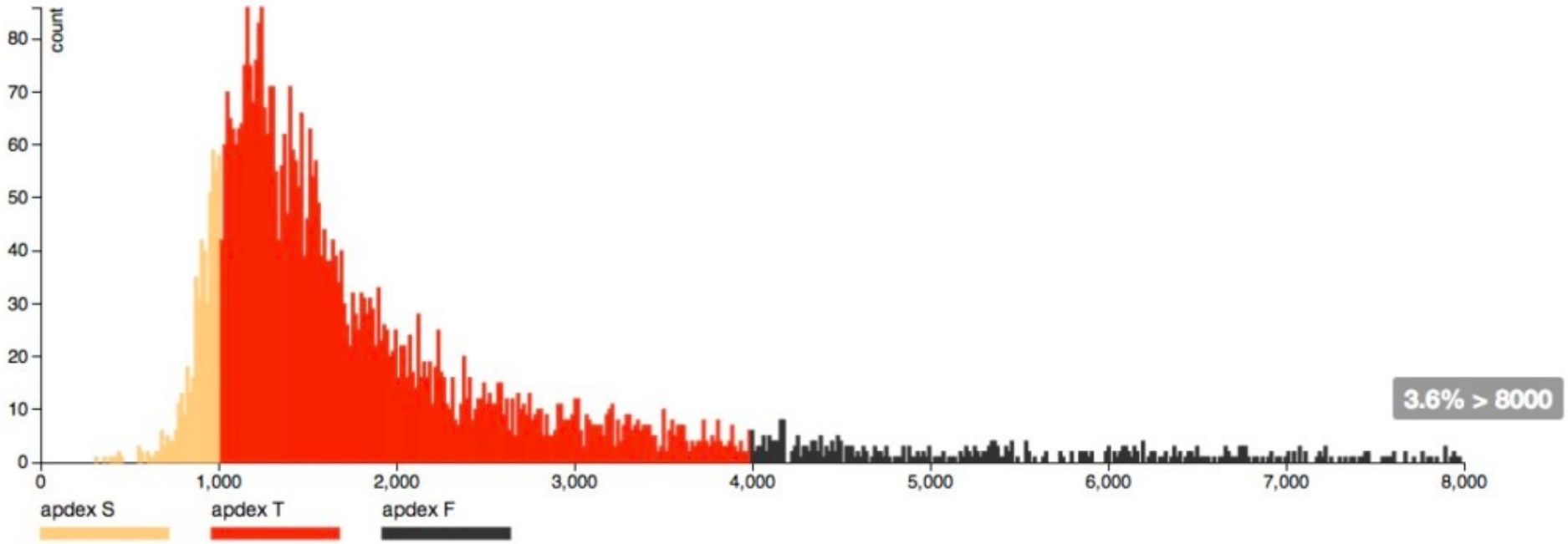
- Given a threshold latency *T* for user satisfaction:
  - *Satisfactory* requests take $t \leq T$
  - *Tolerable* requests take $T < t \leq 4T$
  - *Frustrated* requests task $t > 4T$
  - Apdex = (#satisfactory + 0.5(#tolerable)) / #reqs
  - 0.85 to 0.93 generally “good”
- **Warning!** Can hide *systematic* outliers if not used carefully!
  - e.g. critical action occurs once in every 15 clicks but takes 10x as long => $(14+0)/15 > 0.9$

Apdex Visualization



T=1500ms, Apdex = 0.7

Apdex Visualization



T=1000ms, Apdex = 0.49

What to do if site is slow?

- Small site: overprovision
  - applies to presentation & logic tier
  - before cloud computing, this was painful
  - today, it's largely automatic
- Large site: worry?
  - Provision 1,000-computer site by 10% = 100 idle computers
- Auto-scaling

Continuous Integration & Continuous Deployment

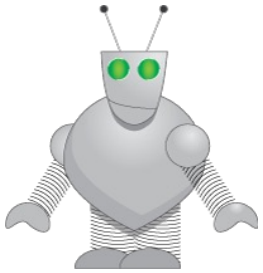
Releases Then and Now

- Facebook: master branch pushed once a week, aiming for once a day (Bobby Johnson, Dir. of Eng., in late 2011)
- Amazon: several deploys per week
- StackOverflow: multiple deploys per day (Jeff Atwood, co-founder)
- GitHub: tens of deploys per day (Zach Holman)
- *Rationale:* **risk** == *#* of engineer-hours invested in product since last deploy!

  Like development and feature check-in, deployment should be a ***non-event*** *that happens all the time*

*** Tesla's 'Full Self-Driving' software is starting to roll out to select customers. October 21, 2020.**

Successful Deployment



- **Automation:** *consistent* deploy process

- **Continuous integration:** integration-testing the app beyond what each developer does
  - Pre-release code checkin triggers CI
  - Since frequent checkins, CI always running
  - Common strategy: integrate with GitHub

Why CI?

- Differences between dev & production envs
- Cross-browser or cross-version testing
- Testing SOA integration when remote services act wonky
- Hardening: protection against attacks
- Stress testing/longevity testing of new features/code paths
- Example: Salesforce CI runs 150K+ tests (number obtained a few years ago) and automatically opens bug report when test fails

Upgrades & Feature Flags

The trouble with upgrades

- What if upgraded code is rolled out to **many** servers?
  - During rollout, some will have version *n* and others version *n*+1…will that work?
- What if upgraded code goes with schema migration?
  - Schema version *n*+1 breaks current code
  - New code won't work with current schema

Naïve update

1. Take service offline
2. Apply destructive migration, including data copying
3. Deploy new code
4. Bring service back online

- May result in unacceptable downtime

Incremental Upgrades with Feature Flags

1. Do nondestructive migration
2. Deploy method protected by feature flag
3. Flip feature flag on; if disaster, flip it back
4. Once all records moved, deploy new code without feature flag
5. Apply migration to remove old columns

Schema change: name => family name, given name

- Create a migration that makes only those changes to the schema that add new tables or columns including a column indicating whether the current record has been migrated to the new schema or not
- Create version n+1 of the app in which every code path affected by the schema change is split into two code paths, of which one or the other is executed based on the value of a feature flag

- Deploy version n+1, which may require pushing the code to multiple servers
- Once deployment is complete, while the app is running set the feature flag's value to True
  - Each record will be migrated to the new schema the next time it's modified for any reason
  - Low-traffic background job that migrates records
- If something goes wrong, turn off the feature flag and revert to version n

- If all goes well, once all records have been migrated, deploy code version n+2, in which the feature flag is removed and only the code path associated with the new schema remains
- Finally, apply a new migration that removes the old name column and the temporary migrated column

“Undoing” an upgrade

- Disaster strikes…use down-migration?
  - is it thoroughly tested?
  - is migration reversible?
  - are you sure someone else didn’t apply an irreversible migration?

- Use feature flags instead
  - Down-migrations are primarily for *development*

Other uses for feature flags

- Preflight checking: gradual rollout of a feature to increasing numbers of users
  - to scope for performance problems, e.g.
- A/B testing
- Complex feature whose code spans multiple deploys

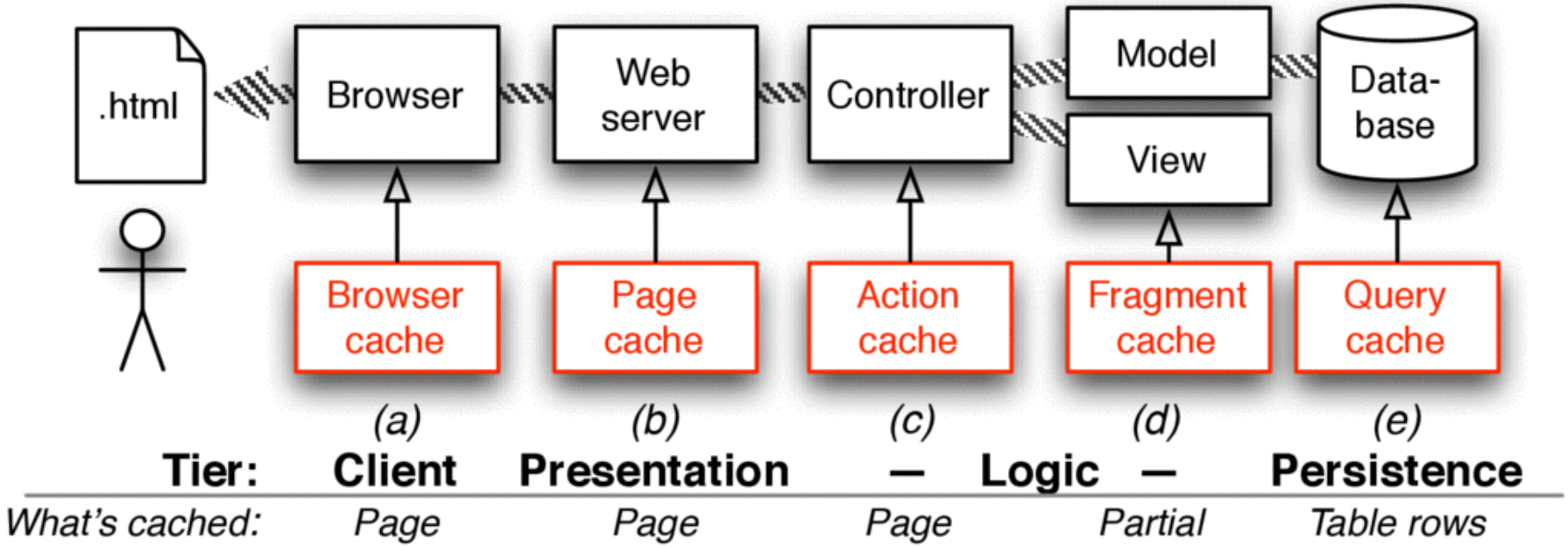
Caching: Improving Rendering Time & Database Performance

The fastest database is the one you don’t use

- **Caching:** Avoid touching database if answer to a query hasn’t changed

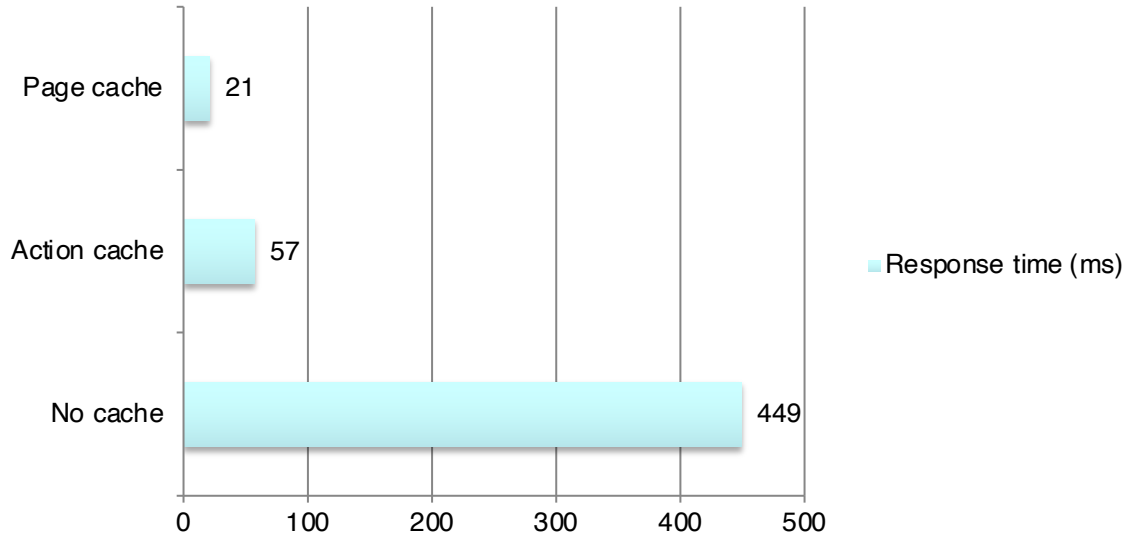
1. Identify what to cache
2. Invalidate (get rid of) *stale* cached versions when underlying DB changes
3. Evict when the cache becomes full

Cache flow


.html
Browser
Web server
Controller
Model
View
Data-base
Browser cache
Page cache
Action cache
Fragment cache
Query cache
(a)
(b)
(c)
(d)
(e)
Tier: Client Presentation — Logic — Persistence
What's cached: Page Page Page Partial Table rows

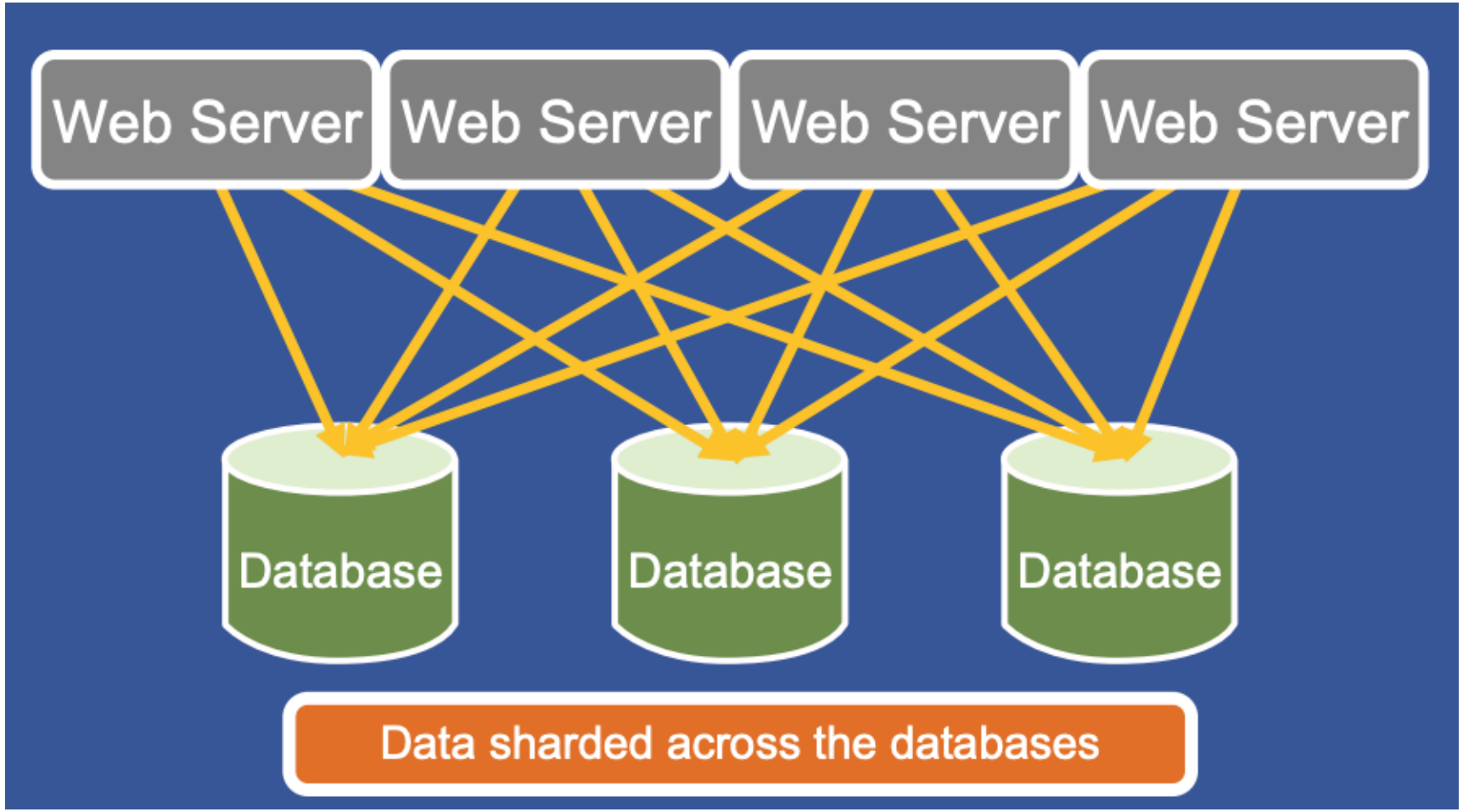
How much does caching help?

- With ~1K movies and ~100 reviews/movie in an app on Heroku, `heroku logs` shows:



- Can serve 8x to 21x as many users with same number of servers if caching used

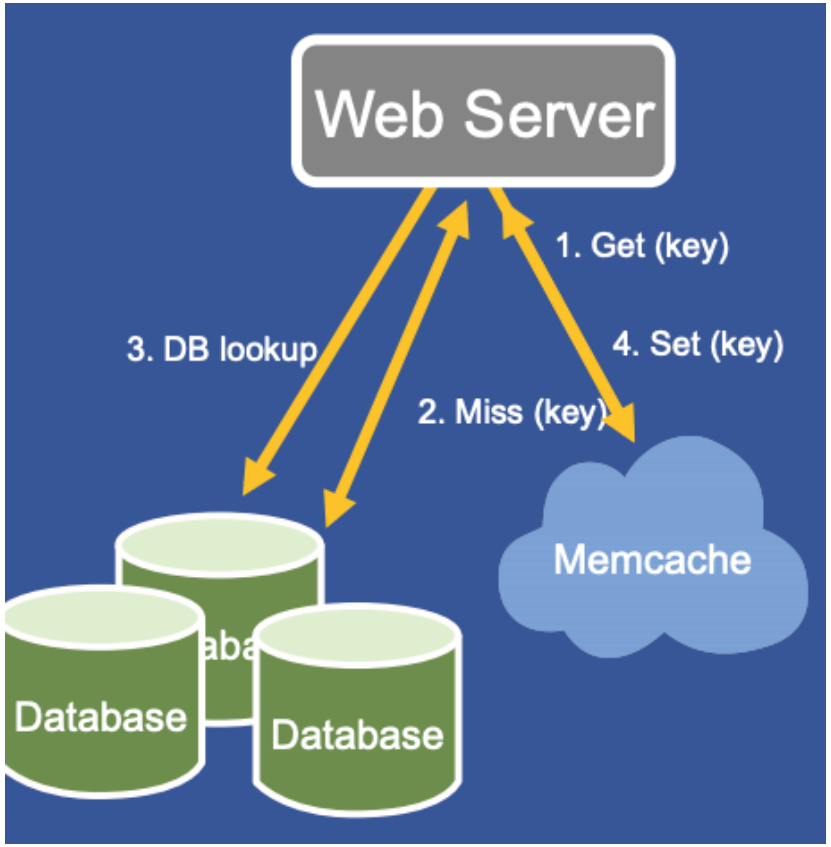
Pre-Memcache



Scaling Memcache at Facebook, NSDI 2013

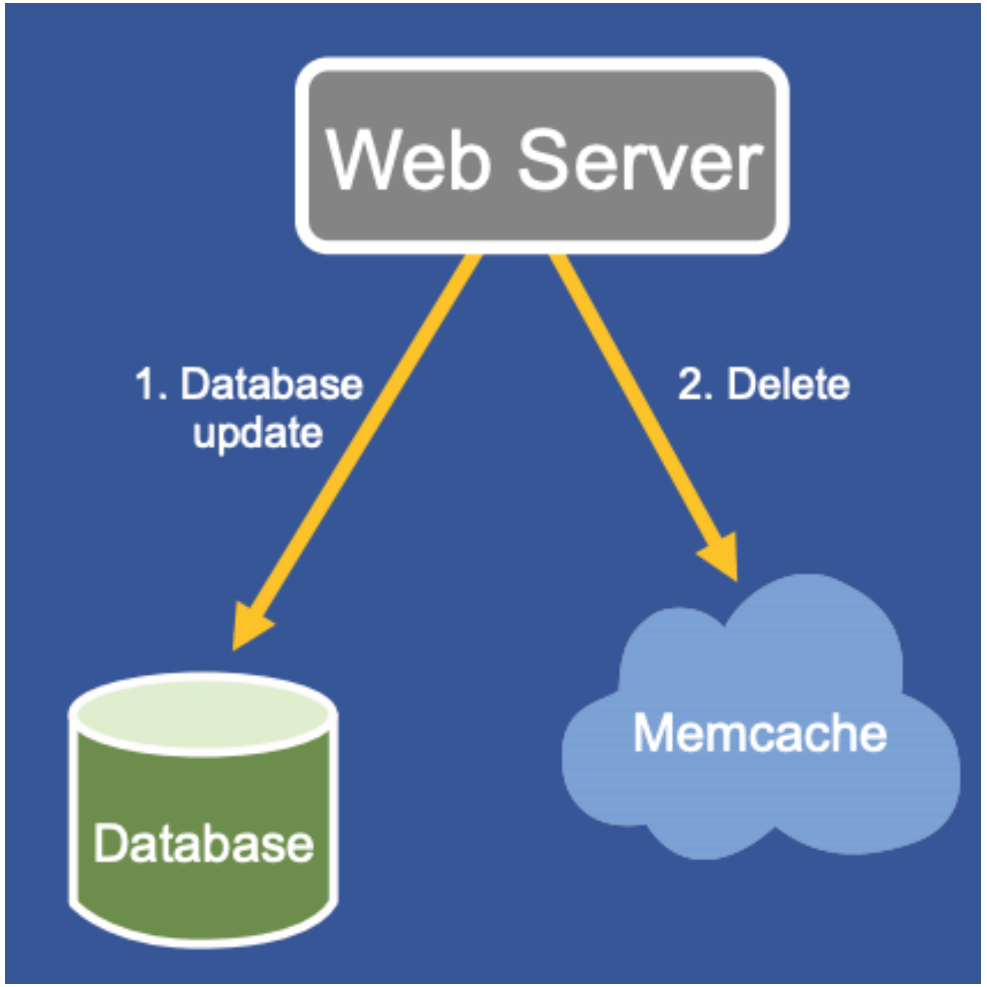
Needs More Read Capacity

- Two orders of magnitude more reads than writes
- Solution: Deploy a few memcache hosts to handle the read capacity
- How do we store data?
  - Demand-filled look-aside cache
  - Common case is data is available in the cache

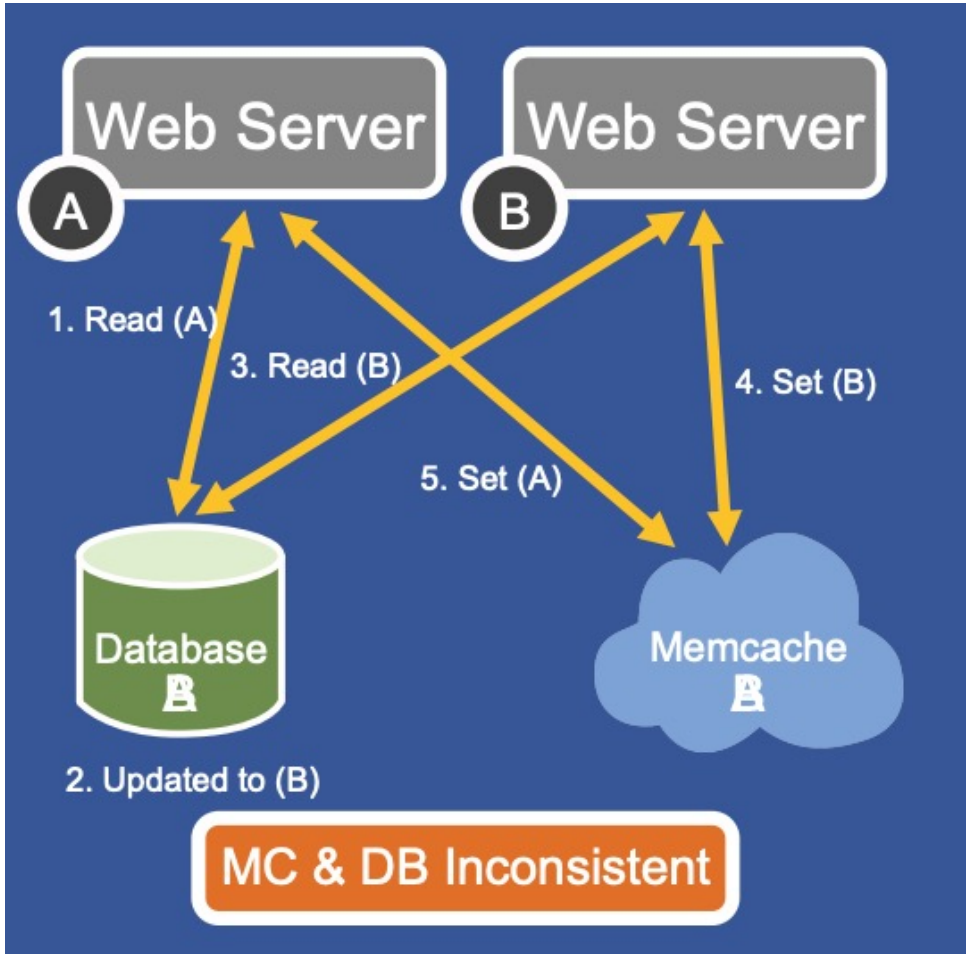


Handling updates

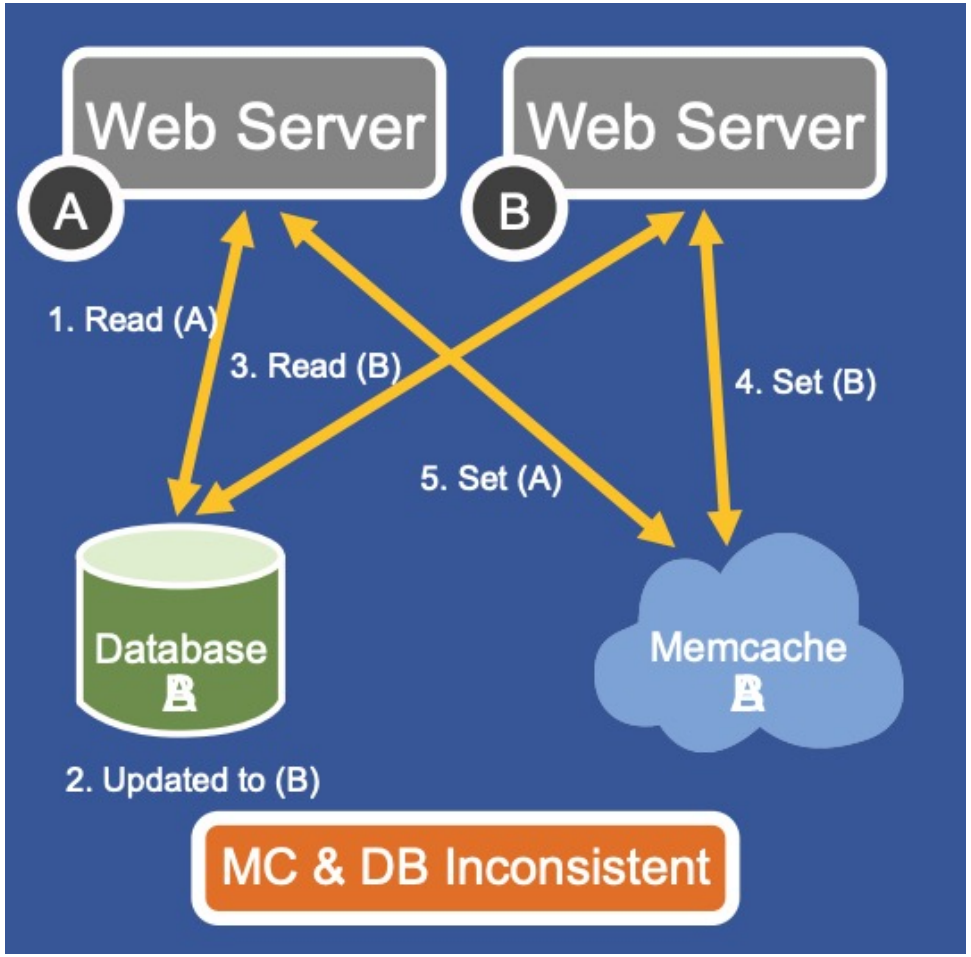
- Memcache needs to be invalidated after DB write
- Prefer deletes to sets
  - Idempotent
  - Demand filled
- Up to web application to specify which keys to invalidate after database update



Problems with look-aside caching: Stale Sets

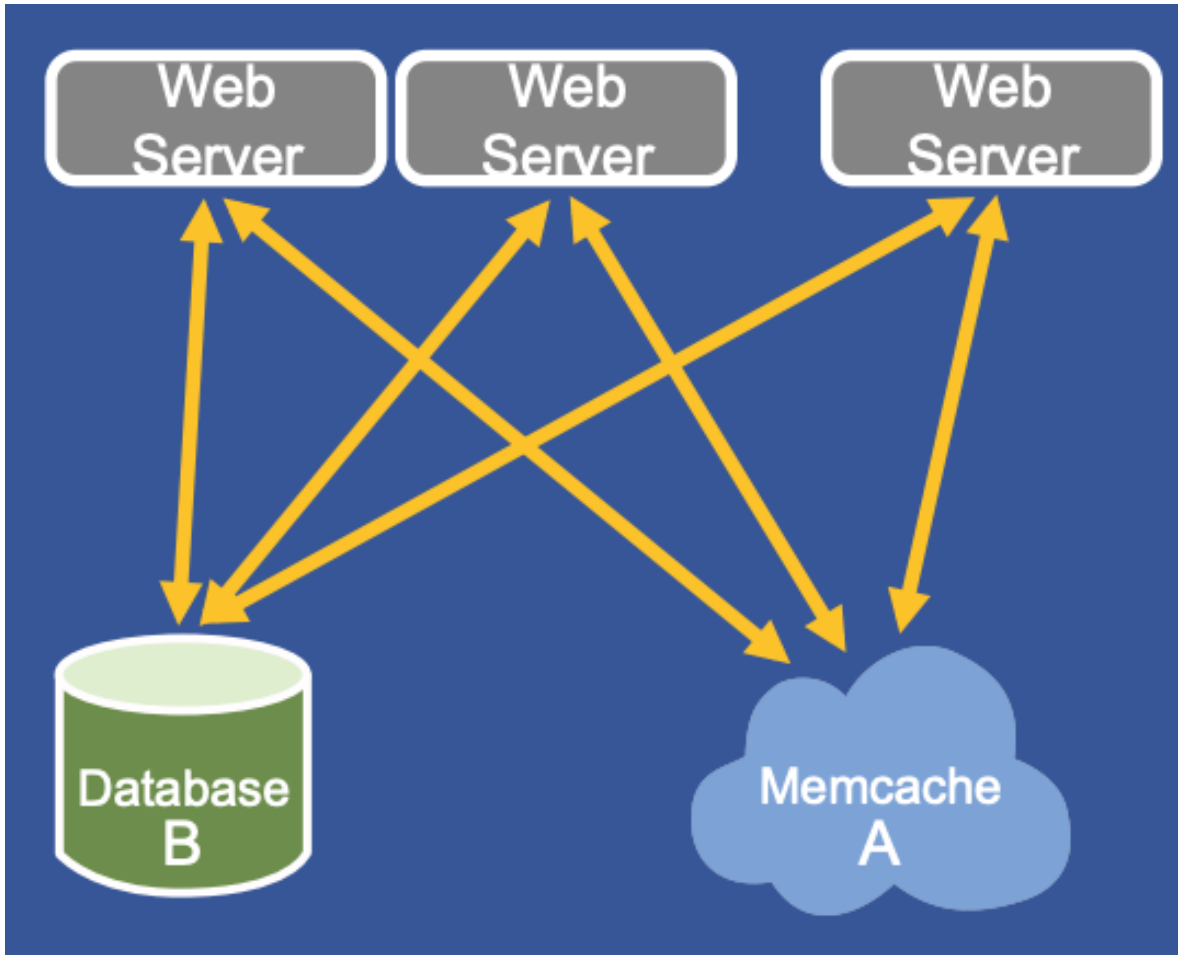


Problems with look-aside caching: Stale Sets

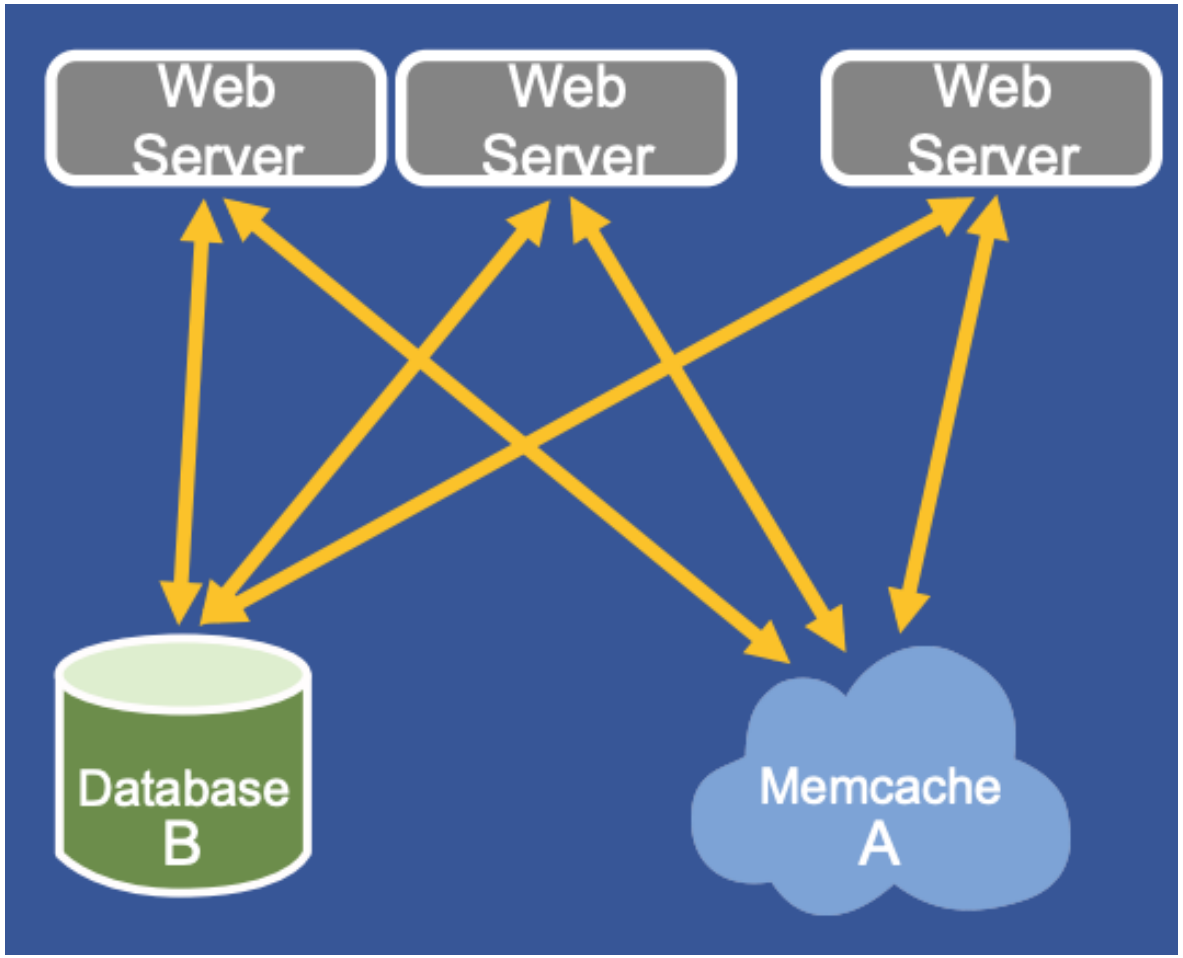


- Extend memcache protocol with “leases”
  - Return and attach a lease-id with every miss
  - Lease-id is invalidated inside server on a delete
  - Disallow set if the lease-id is invalid at the server

Problems with look-aside caching: Thundering Herds

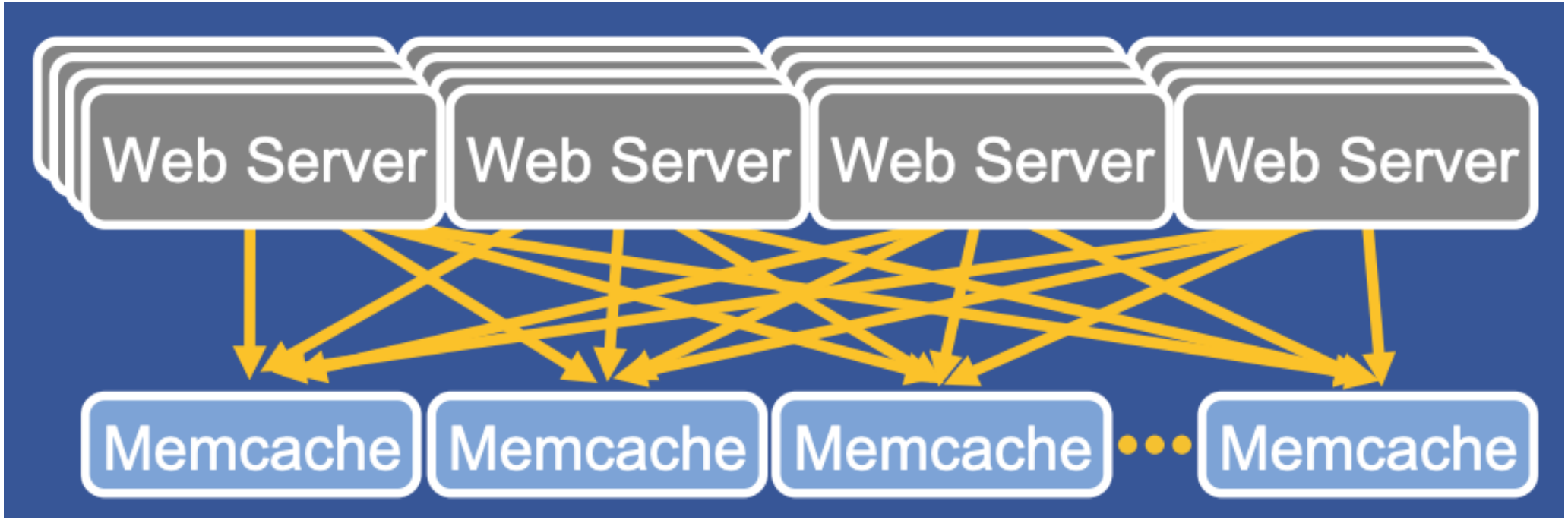


Problems with look-aside caching: Thundering Herds



- Memcache server arbitrates access to database
  - Small extension to leases
- Clients given a choice of using a slightly stale value or waiting

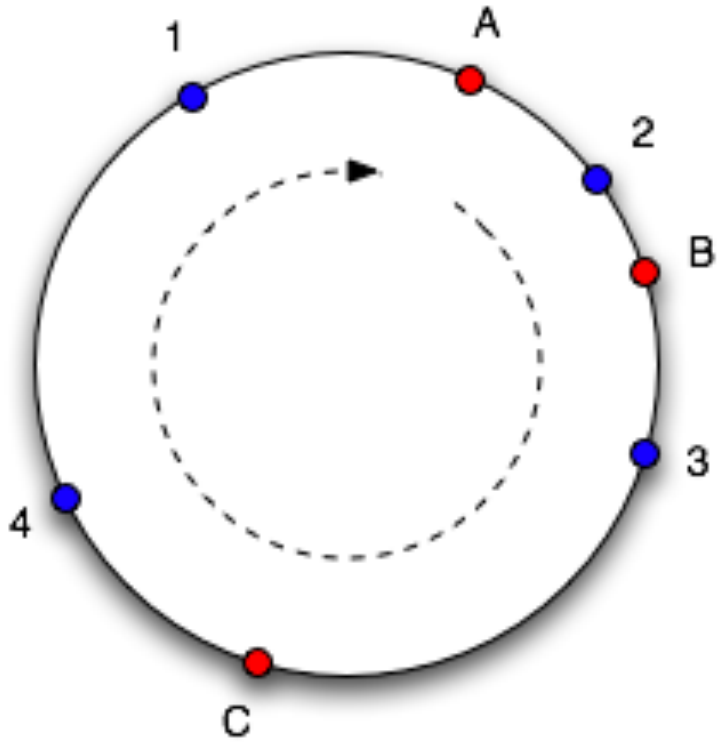
Need even more read capacity



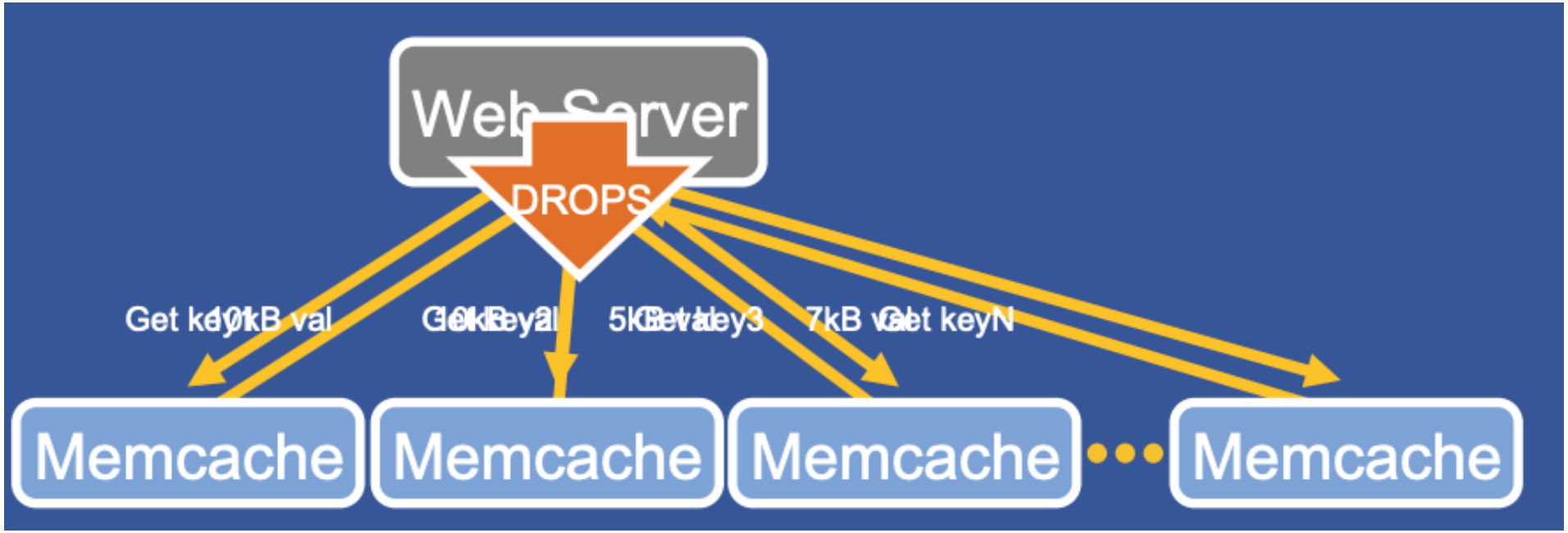
- Items are distributed across memcache servers by using **consistent hashing** on the key
- All web servers talk to all memcache servers
  - Accessing 100s of memcache servers to process a user request is common

Consistent Hashing

- Hash both objects and caches (nodes) using the same hash function
- Map the cache to an interval, which will contain a number of object hashes.
- If the cache is removed then its interval is taken over by a cache with an adjacent interval.
  All the other caches remain unchanged.



Incast congestion



- Many simultaneous responses overwhelm shared networking resources
- Solution: Limit the number of outstanding requests with a sliding window

Avoiding Abusive Queries

Be kind to the database

- Outgrowing single-machine database == big investment: sharding, replication, etc.
- Alternative: find ways to relieve pressure on database so can stay in “PaaS-friendly” tier
  1. Use caching to reduce number of database accesses
  2. Avoid “n+1 queries” problem in association
  3. Use indices judiciously

n+1 queries problem

- **Problem:** you are doing n+1 queries to traverse an association, rather than 1 query
  (a common performance anti-pattern)
- E.g., you need to iterate through all the cars, and for each one, print out a list of the wheels. The naive O/R implementation

  SELECT * FROM Cars;
  for each Car:
  SELECT * FROM Wheel WHERE CarId = ?

  one select for the Cars, and then N additional selects, where N is the total number of cars.

n+1 queries problem

- select_related: django to do a join when fetching data.
  You should use it on any lookup where you know you'll need related fields.

  e = Entry.objects.get(id=5) # Hits the db
  b = e.blog # Hits the db again

  => e = Entry.objects.select_related('blog').get(id=5) # Hits the db
  b = e.blog # Doesn't hit the db

```
class Blog(models.Model):
…
class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
…
```

Table Scan Solved by Indices

- Speeds up access when searching DB table by column other than primary key
- Similar to using a hash table
  - alternative is *table scan*—bad!;
    Taking time O(n) for a table with n rows
  - even bigger win if attribute is unique-valued
- Why not index *every* column?
  - takes up space
  - all indices must be updated when table updated

What to index?

- Foreign key columns
- Columns that appear in `where()` clauses
- Columns on which you sort

How much does indexing help?

(Numbers depend on environments, workloads, etc.)

# of reviews:	2000	20,000	200,000
Read 100, no indices	0.94	1.33	5.28
Read 100, FK indices	0.57	0.63	0.65
Performance	166%	212%	808%

Database as a Service

- Access to a database without the need for setting up physical hardware, installing software or configuring for performance

- Autoscaling (hopefully) handled

Availability, Consistency, Partition Tolerance

Fault Assumption

- Fail-stop failure
  - When a node fails, it ceases to function entirely.
  - May resume normal operation when restarted.
- Byzantine failure

- Messages
  - May be lost.
  - May be duplicated.
  - May be delayed (and thus reordered).
  - May ***not*** be corrupt.

Ways to Improve the Reliability of Software

- Overprovisioning deals gracefully with server crashes
  - Temporarily losing one server out of n servers degrades performance by 1/n
  - An easy solution is to overprovision by deploying n+1 servers
  - At large scale, systematic overprovisioning is infeasible

- Make it more robust

Fault Tolerance

- Redundancy
- Storage: redundant copies
  - Immutable data: replication, erasure coding
  - Mutable data: quorum systems
  - Database: distributed transactions
- Computation: state machine replication (consensus algorithm)
  - Paxos
    The Part-Time Parliament (Leslie Lamport)
    https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
- CAP theorem

Replication, Erasure Coding

- Replication: create k identical copies of a data block. We can read a data block by reading any one copy.
  - Tolerate k-1 failures
  - Recovery: in case of a replica failure, create a new copy to maintain k replicas

- Erasure coding (m, k): a data block is coded into m fragments. We can read a data block by using any k fragments out of m fragments ($k < m$).
  - Tolerate m-k failures
  - Recovery: in case of a fragment failure, re-generate a fragment

Quorum System

- Quorum protocol: 2f+1 (R/W) replicas to tolerate f failures
  - Read protocol
  - Write protocol
- Consistency tradeoff: e.g., Dynamo (AWS)

CAP Theorem

- Conjectured by Prof. Eric Brewer at PODC (Principle of Distributed Computing) 2000 keynote talk

- Described the *trade-offs involved in distributed system*

- It is impossible for a web service to provide following *three guarantees at the same time*:
  - **Consistency**
  - **Availability**
  - **Partition-tolerance**

CAP Theorem

- **C**onsistency:
  - All nodes should see the same data at the same time
- **A**vailability:
  - Node failures do not prevent survivors from continuing to operate
- **P**artition-tolerance:
  - The system continues to operate despite network partitions
- A distributed system can satisfy any two of these guarantees at the same time **but not all three**

Why this is important?

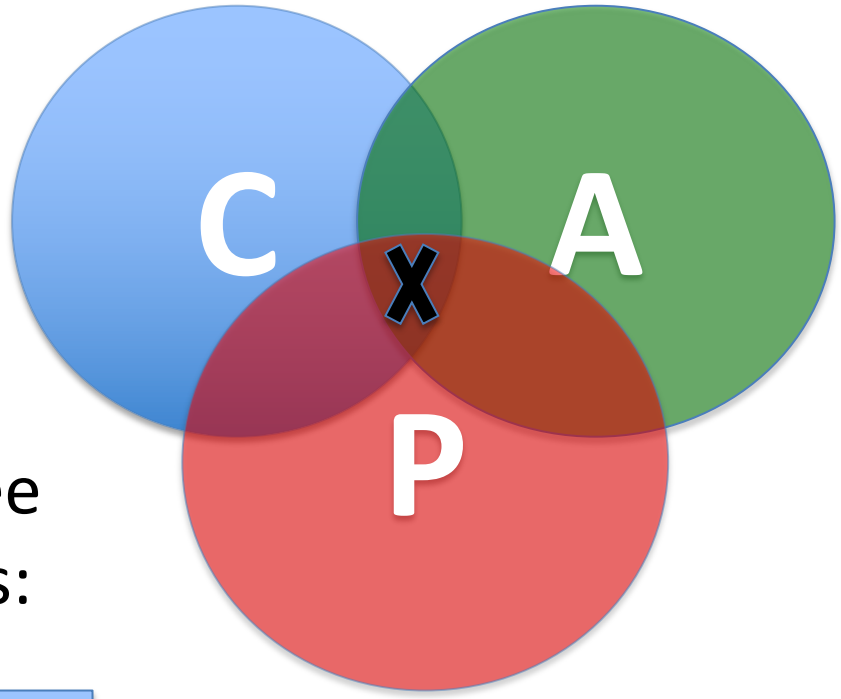
- Large-scale systems are **distributed** (Big Data Trend, etc.)
- CAP theorem describes the **trade-offs** involved in distributed systems
- A proper understanding of CAP theorem is essential to **making decisions** about the future of distributed system **design**
- Misunderstanding can lead to **erroneous or inappropriate** design choices

Problem for ”Relational Database” to Scale

- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which unfortunately is **impossible** …

Revisit CAP Theorem

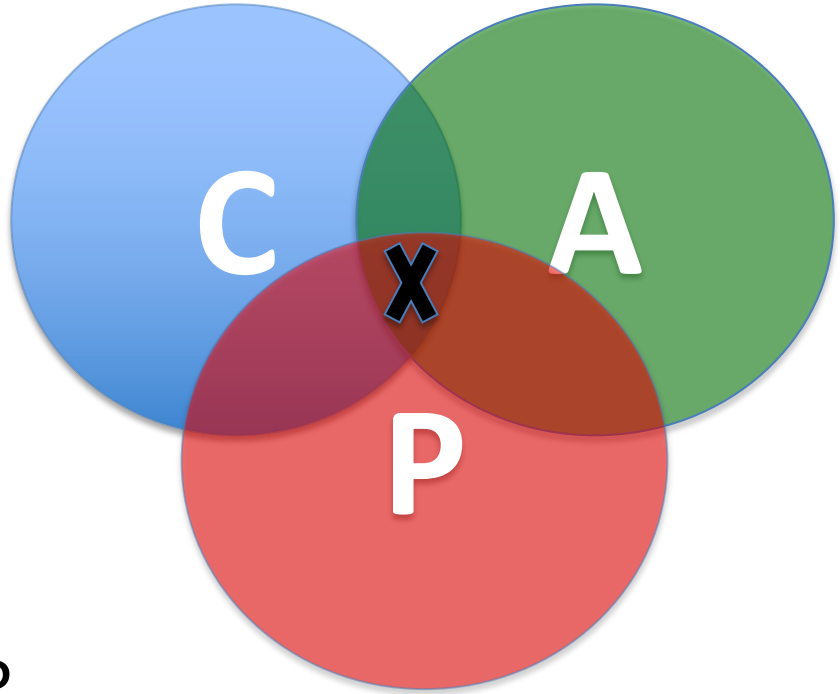
- Of the following three guarantees potentially offered a by distributed systems:
  - Consistency
  - Availability
  - Partition tolerance
- Pick two
- This suggests there are three kinds of distributed systems:
  - CP
  - AP
  - CA



Any problems?

Consistency or Availability

- Consistency and Availability is not “binary” decision
- AP systems relax consistency in favor of availability – but are not inconsistent
- CP systems sacrifice availability for consistency- but are not unavailable
- This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



AP: Best Effort Consistency

- Example:
  - Web Caching
  - DNS
- Trait:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

CP: Best Effort Availability

- Example:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Trait:
  - Pessimistic locking
  - Make minority partition unavailable

Types of Consistency

- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- **Eventual Consistency**
  - Specific form of weak consistency
  - It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)

Eventual Consistency Variations

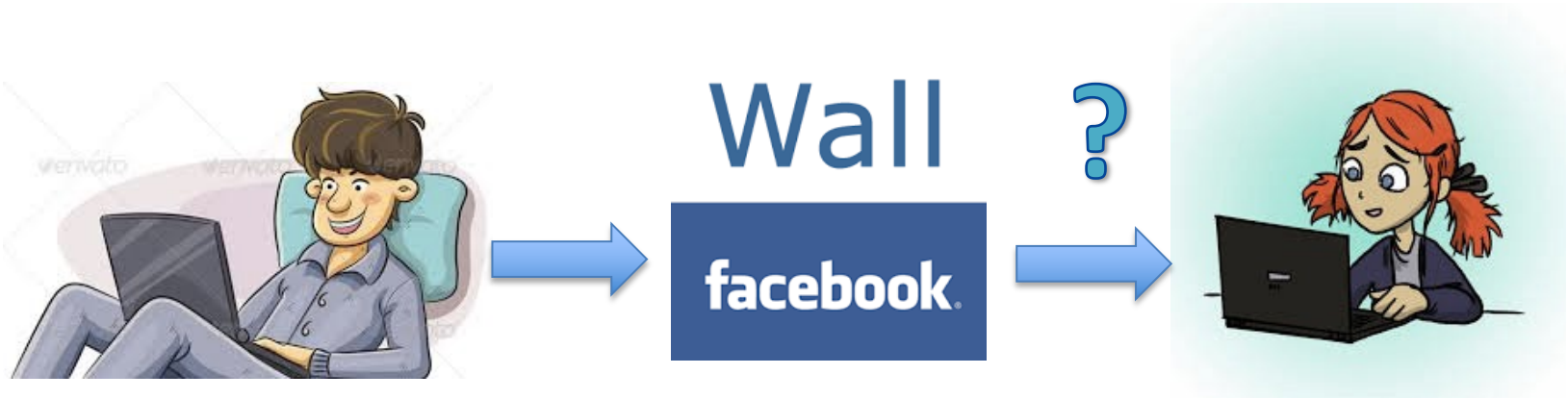
- Causal consistency
  - Processes that have causal relationship will see consistent data
- Read-your-write consistency
  - A process always accesses the data item after it's update operation and never sees an older value
- Session consistency
  - As long as session exists, system guarantees read-your-write consistency
  - Guarantees do not overlap sessions

Eventual Consistency Variations

- Monotonic read consistency
  - If a process has seen a particular value of data item, any subsequent processes will never return any previous values
- Monotonic write consistency
  - The system guarantees to serialize the writes by the *same* process
- In practice
  - A number of these properties can be combined
  - Monotonic reads and read-your-writes are most desirable

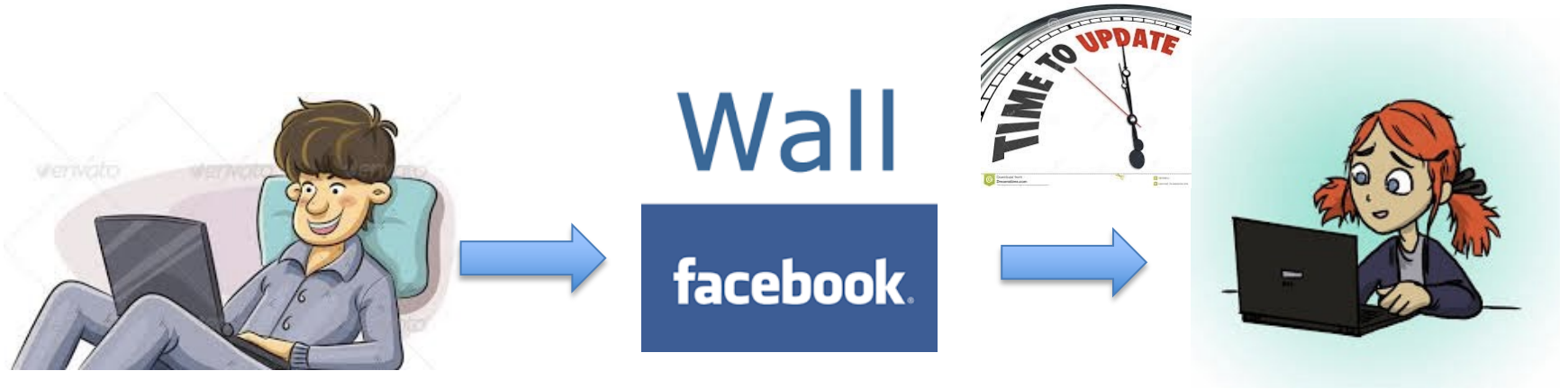
Eventual Consistency
- A Facebook Example

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:
  - **Nothing is there!**



Eventual Consistency
- A Facebook Example

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:
  - **She finds the story Bob shared with her!**



Eventual Consistency - A Facebook Example

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than multi-billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

Paxos: The Part-Time Parliament

- Parliament determines laws by passing sequence of numbered decrees
- Legislators can leave and enter the chamber at arbitrary times
- No centralized record of approved decrees– instead, each legislator carries a ledger

Paxos Requirements

- Safety
  - Only a value that has been proposed may be chosen
  - Only a single value is chosen
  - A process never learns that a value has been chosen until it actually has been
- Liveness
  - Some proposed value is eventually chosen.
  - If a value has been chosen, a node can eventually learn the value

Paxos Notation

- Classes of agents:
  - Proposers
  - Acceptors
  - Learners

- A node can act as more than one clients (usually three)

Paxos Algorithm

- Phase 1 (prepare):
  - A proposer selects a proposal number n and sends a *prepare request* with number n to majority of acceptors.
  - If an acceptor receives a prepare request with number n greater than that of any prepare request it saw, it responses YES to that request with a promise not to accept any more proposals numbered less than n and include the highest-numbered proposal (if any) that it has accepted.

Paxos Algorithm

- Phase 2 (accept):
  - If the proposer receives a response YES to its prepare requests from a majority of acceptors, then it sends an *accept request* to each of those acceptors for a proposal numbered n with a values *v* which is the value of the highest-numbered proposal among the responses.
  - If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

* A value is chosen at proposal number n iff majority of acceptor accept that value in phase 2 of the proposal number.

Make it More Robust

- Defensive programming: anticipate potential software flaws and write code to handle them

- Examples
  - Check input values
  - Check input data type
  - Catch exceptions

Longevity Bugs

- Resource leak (RAM, file buffers, sessions table) is a classic example

- Some infrastructure software such as Apache web server already does *rejuvenation*
  - aka “rolling reboot”

Disaster Recovery

- Test the system still works even if a cluster or an entire data center is lost

- Facebook Project Storm

Monitoring

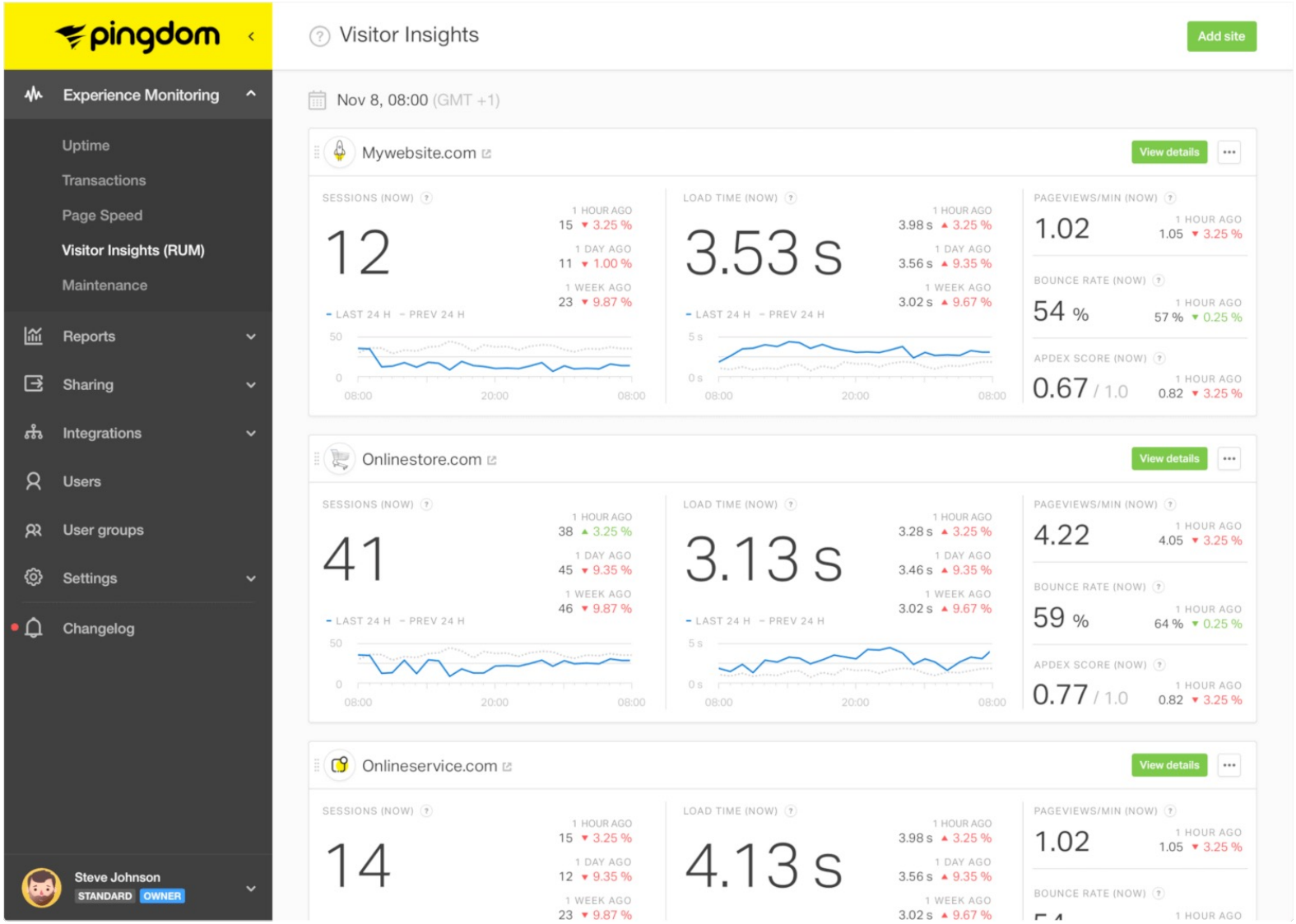
Kinds of monitoring

- **“If you’re not monitoring it, it’s probably broken”**
- At development time (*profiling*)
  - Identify possible performance/stability problems *before* they get to production
- In production
  - Internal: instrumentation embedded in app and/or framework
  - External: active probing by other site(s).

Why use external monitoring?

- Detect if site is down
- Detect if site is slow for reasons outside measurement boundary of internal monitoring
- Get user’s view from many different places on the Internet
- Example: Pingdom

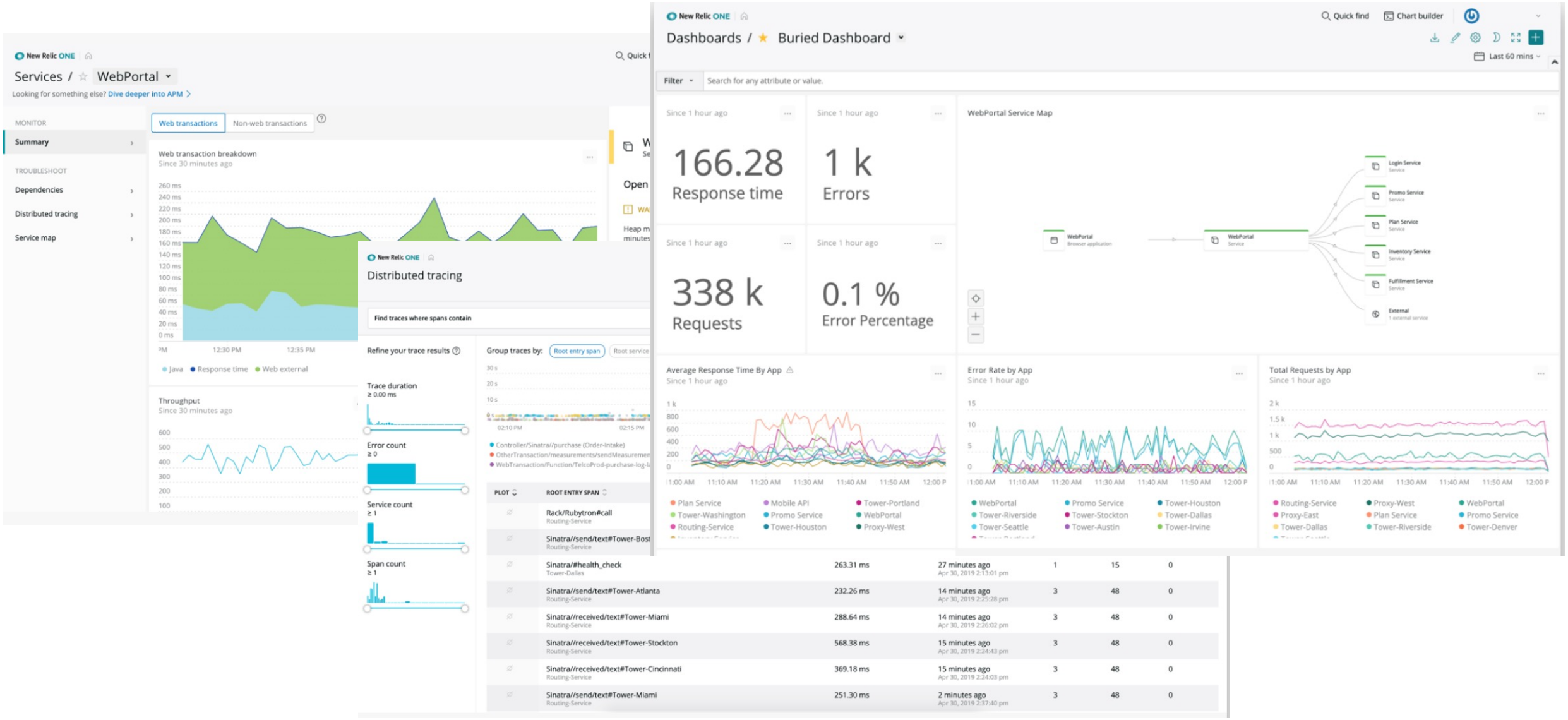
pingdom

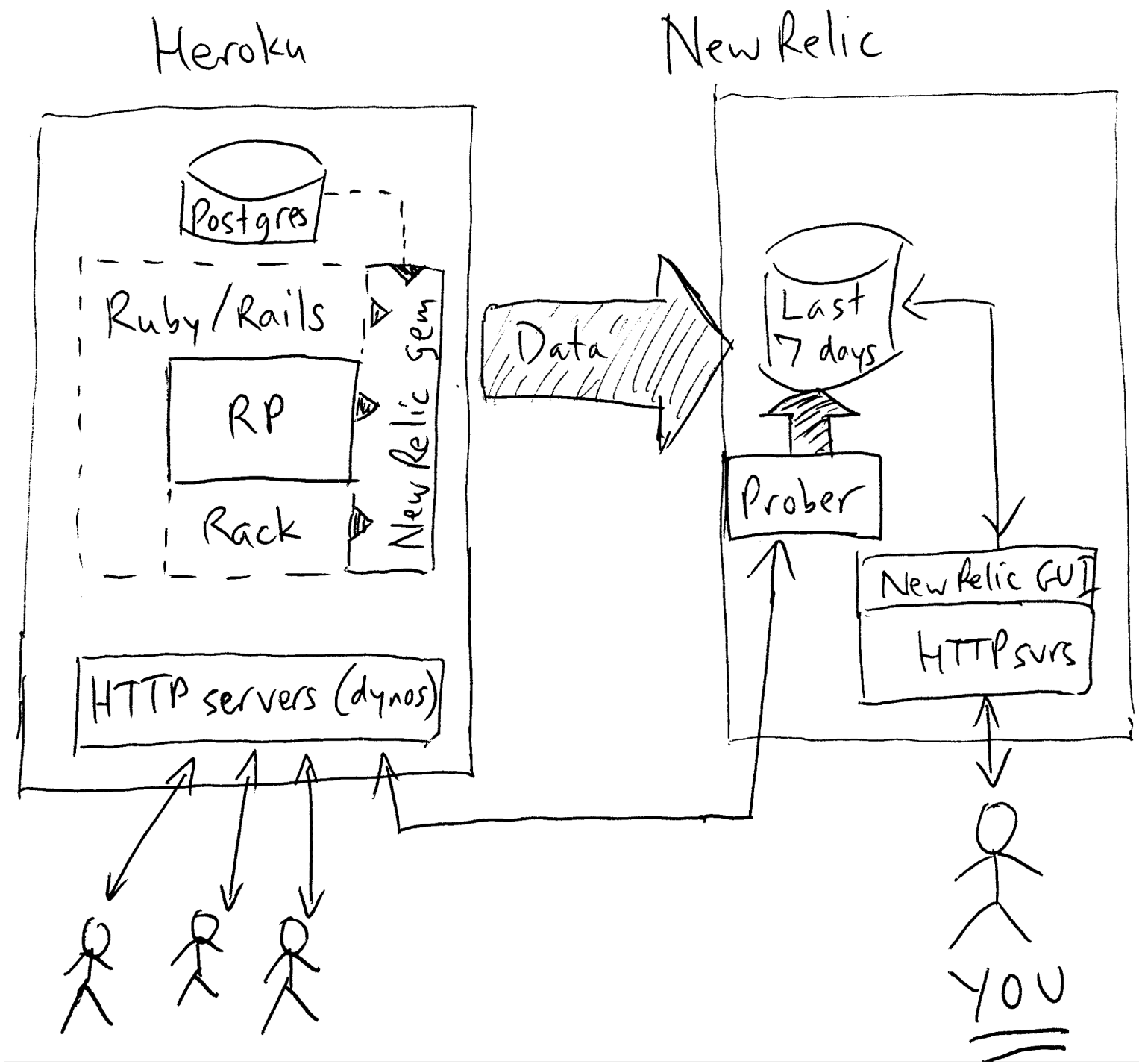

pingdom
Visitor Insights
Add site
Experience Monitoring
Uptime
Transactions
Page Speed
Visitor Insights (RUM)
Maintenance
Reports
Sharing
Integrations
Users
User groups
Settings
Changelog
Steve Johnson
STANDARD OWNER
Nov 8, 08:00 (GMT +1)
Mywebsite.com
View details
SESSIONS (NOW)
12
1 HOUR AGO 15 ▼ 3.25 %
1 DAY AGO 11 ▼ 1.00 %
1 WEEK AGO 23 ▼ 9.87 %
LOAD TIME (NOW)
3.53 s
1 HOUR AGO 3.98 s ▲ 3.25 %
1 DAY AGO 3.56 s ▲ 9.35 %
1 WEEK AGO 3.02 s ▲ 9.67 %
PAGEVIEWS/MIN (NOW)
1.02
1 HOUR AGO 1.05 ▼ 3.25 %
BOUNCE RATE (NOW)
54 %
1 HOUR AGO 57 % ▼ 0.25 %
APDEX SCORE (NOW)
0.67 / 1.0
1 HOUR AGO 0.82 ▼ 3.25 %
LAST 24 H PREV 24 H
08:00 20:00 08:00
Onlinestore.com
View details
SESSIONS (NOW)
41
1 HOUR AGO 38 ▲ 3.25 %
1 DAY AGO 45 ▼ 9.35 %
1 WEEK AGO 46 ▼ 9.87 %
LOAD TIME (NOW)
3.13 s
1 HOUR AGO 3.28 s ▲ 3.25 %
1 DAY AGO 3.46 s ▲ 9.35 %
1 WEEK AGO 3.02 s ▲ 9.67 %
PAGEVIEWS/MIN (NOW)
4.22
1 HOUR AGO 4.05 ▼ 3.25 %
BOUNCE RATE (NOW)
59 %
1 HOUR AGO 64 % ▼ 0.25 %
APDEX SCORE (NOW)
0.77 / 1.0
1 HOUR AGO 0.82 ▼ 3.25 %
Onlineservice.com
View details
SESSIONS (NOW)
14
1 HOUR AGO 15 ▼ 3.25 %
1 DAY AGO 12 ▼ 9.35 %
1 WEEK AGO 23 ▼ 9.87 %
LOAD TIME (NOW)
4.13 s
1 HOUR AGO 3.98 s ▲ 3.25 %
1 DAY AGO 3.56 s ▲ 9.35 %
1 WEEK AGO 3.02 s ▲ 9.67 %
PAGEVIEWS/MIN (NOW)
1.02
1 HOUR AGO 1.05 ▼ 3.25 %
BOUNCE RATE (NOW)

Internal monitoring

- pre-SaaS/PaaS: *local*
  - Info collected & stored locally, e.g., Nagios
- Today: *hosted*
  - Info collected in your app but stored centrally
  - Info available even when app is down
- Example: Facebook ODS, New Relic
  - conveniently, has both a development mode and production mode

New Relic


New Relic ONE
Services / WebPortal
Looking for something else? Dive deeper into APM
MONITOR
Summary
TROUBLESHOOT
Dependencies
Distributed tracing
Service map
Web transactions
Non-web transactions
Web transaction breakdown
Since 30 minutes ago
Java
Response time
Web external
Throughput
Since 30 minutes ago
Distributed tracing
Find traces where spans contain
Refine your trace results
Trace duration
Error count
Service count
Span count
Group traces by:
Root entry span
PLOT
ROOT ENTRY SPAN
Rack/Rubytron#call
Sinatra//send/text#Tower-Bost
Sinatra/#health_check
263.31 ms
Sinatra//send/text#Tower-Atlanta
232.26 ms
Sinatra//received/text#Tower-Miami
288.64 ms
Sinatra//received/text#Tower-Stockton
568.38 ms
Sinatra//received/text#Tower-Cincinnati
369.18 ms
Sinatra//send/text#Tower-Miami
251.30 ms
Dashboards / Buried Dashboard
Quick find
Chart builder
Last 60 mins
Filter
Search for any attribute or value.
Since 1 hour ago
166.28
Response time
1 k
Errors
338 k
Requests
0.1 %
Error Percentage
WebPortal Service Map
Average Response Time By App
Error Rate by App
Total Requests by App
Plan Service
Mobile API
Tower-Portland
Tower-Washington
Promo Service
WebPortal
Routing-Service
Tower-Houston
Proxy-West
Tower-Austin
Tower-Irvine
Tower-Riverside
Tower-Dallas
Tower-Seattle
Proxy-East
Tower-Denver


Heroku
New Relic
Postgres
Ruby/Rails
RP
Rack
New Relic gem
HTTP servers (dynos)
Data
Last 7 days
Prober
New Relic GUI
HTTP svrs
YOU

Sampling of monitoring tools

What is monitored	Level	Example tool	Hosted
Availability	site	pingdom.com	Yes
Slow controller actions or DB queries	app	newrelic.com (also has dev mode)	Yes
Clicks, think times	app	Google Analytics	Yes
Process health & telemetry	process	monit, nagios	No

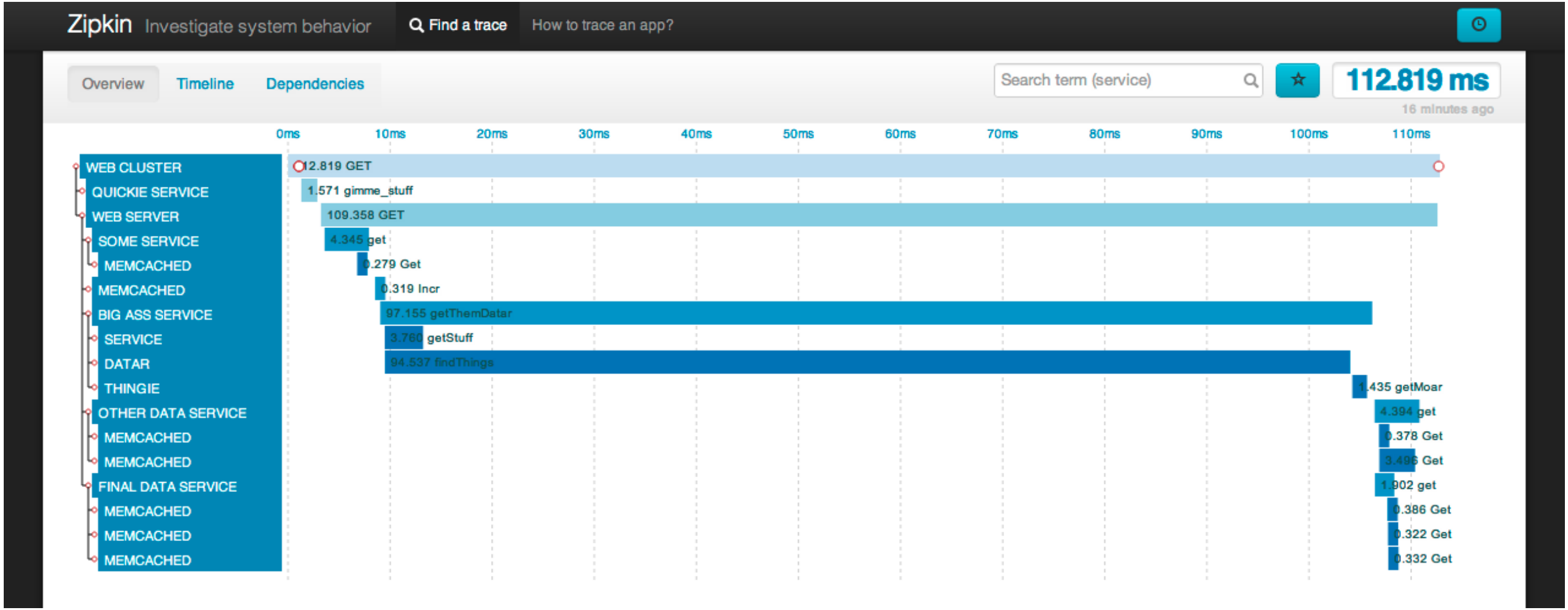
What to measure?

- *Stress testing* or *load testing*: how far can I push my system...
  - ...before performance becomes unacceptable?
  - ...before it gasps and dies?
- Usually, one component will be *bottleneck*
  - a particular view, action, query, …
- Load testers can be simple or sophisticated
  - request on a single URI over and over
  - do a fixed sequence of URI's over and over
  - play back a log file

Request Tracing

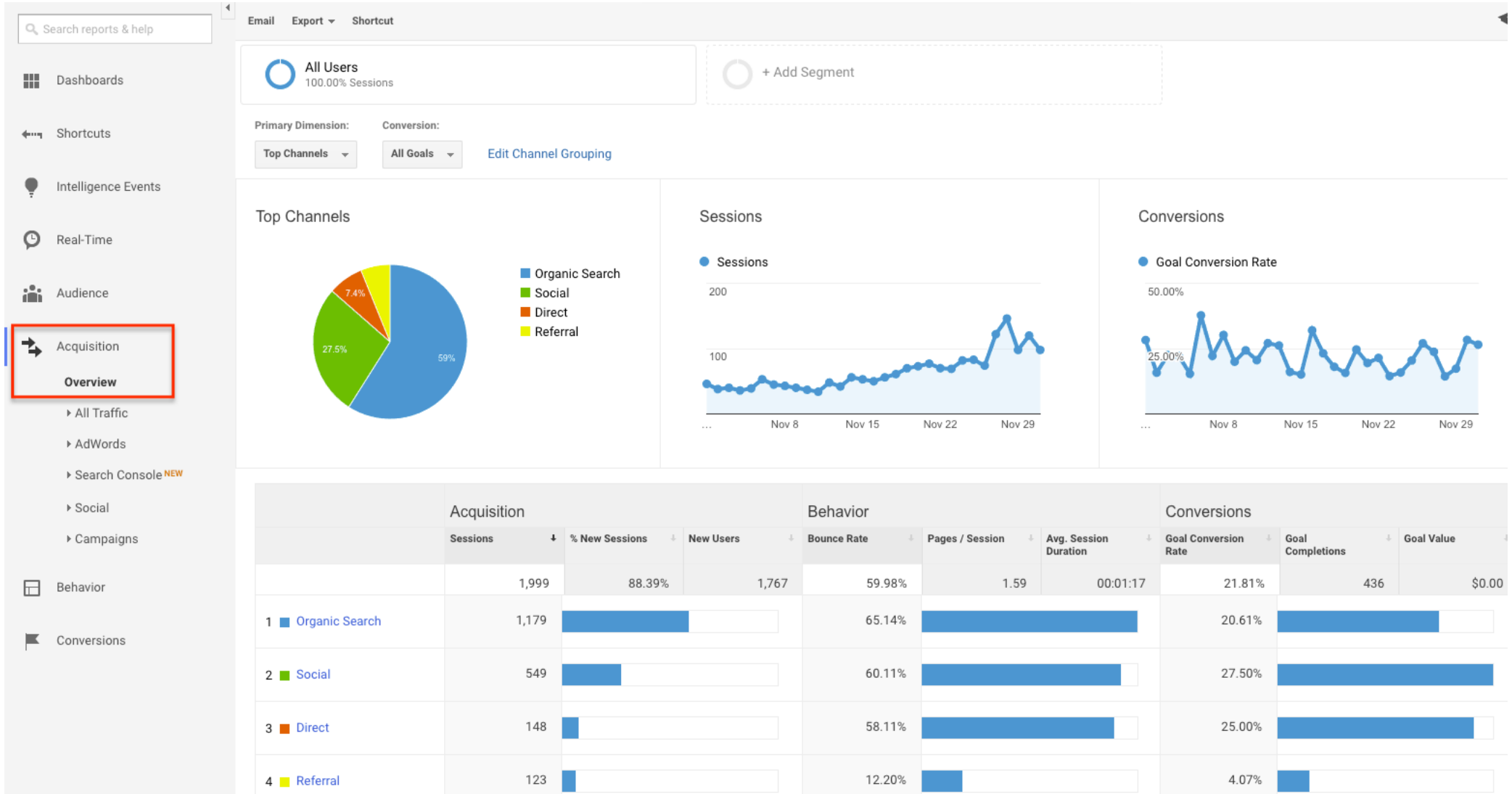
- Typically aggregating metrics such as latency over many requests
- A contrasting approach: request tracing
  - Simplified version of request tracing
    - Db query time, controller action time, rendering time
  - True request tracing is fine grained, following a request through every software component in every tier and timestamping it along the way
    - Google Dapper, Twitter Zipkin, Linkedin Htrace, …

Example: Zipkin



Monitoring for Understanding Customers' Behavior

- Clickstreams: what are the most popular sequences of pages your users visit?
- Dwell times: how long does a typical user stay on a given page?
- Abandonment: if your site contains a flow that has a well-defined termination, such as making a sale, what percentage of users "abandon" the flow rather than completing it and how far do they get?
- E.g., Google analytics
- Advertising! – Most of Google's Revenue


Search reports & help
Dashboards
Shortcuts
Intelligence Events
Real-Time
Audience
Acquisition
Overview
All Traffic
AdWords
Search Console NEW
Social
Campaigns
Behavior
Conversions
Email
Export
Shortcut
All Users
100.00% Sessions
+ Add Segment
Primary Dimension:
Top Channels
Conversion:
All Goals
Edit Channel Grouping
Top Channels
7.4%
27.5%
59%
Organic Search
Social
Direct
Referral
Sessions
Sessions
200
100
Nov 8
Nov 15
Nov 22
Nov 29
Conversions
Goal Conversion Rate
50.00%
25.00%
Nov 8
Nov 15
Nov 22
Nov 29
Acquisition
Behavior
Conversions
Sessions
% New Sessions
New Users
Bounce Rate
Pages / Session
Avg. Session Duration
Goal Conversion Rate
Goal Completions
Goal Value
1,999 88.39% 1,767 59.98% 1.59 00:01:17 21.81% 436 $0.00
1 Organic Search 1,179 65.14% 20.61%
2 Social 549 60.11% 27.50%
3 Direct 148 58.11% 25.00%
4 Referral 123 12.20% 4.07%