Lecture 21:
Scaling a Web Site

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Acknowledgments: Solomon Boulos, Andreas Sundquist

Announcements

- **List of class final projects**

  `http://www.cs.cmu.edu/~15418/projectlist.html`

- **You are encouraged to keep a log of activities, rants, thinking, findings, on your project web page**
  - It will be interesting for us to read
  - It will come in handy when it comes time to do your writeup
  - Writing clarifies thinking

From last time: synchronization granularity

Python's global interpreter lock:

from: http://wiki.python.org/moin/GlobalInterpreterLock

In CPython, the **global interpreter lock**, or **GIL**, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)

CPython extensions must be GIL-aware in order to avoid defeating threads. For an explanation, see Global interpreter lock.

The GIL is controversial because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems in certain situations. Note that potentially blocking or long-running operations, such as I/O, image processing, and NumPy number crunching, happen *outside* the GIL. Therefore it is only in multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a bottleneck.

However the GIL degrades performance even when it is not a bottleneck. Summarizing those slides: The system call overhead is significant, especially on multicore hardware. Two threads calling a function may take twice as much time as a single thread calling the function twice. The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.

Non-CPython implementations

Jython and IronPython have no GIL and can fully exploit multiprocessor systems.

[Mention place of GIL in StacklessPython.]

Eliminating the GIL

Getting rid of the GIL is an occasional topic on the python-dev mailing list. No one has managed it yet. The following properties are all highly desirable for any potential GIL replacement; some are hard requirements.

What you should know

- How concepts we have discussed in this course relate to designing high performance web sites

- Scale-out parallelism: use many machines
  - How is work distributed onto these machines?
  - Elasticity: how to adapt dynamically to varying load

- Caching: How is reuse and locality identified and exploited to avoid redundant computation and/or data-transfer

Our focus today: performance scaling

- Today I'm going to focus on performance issues
  - parallelism and locality

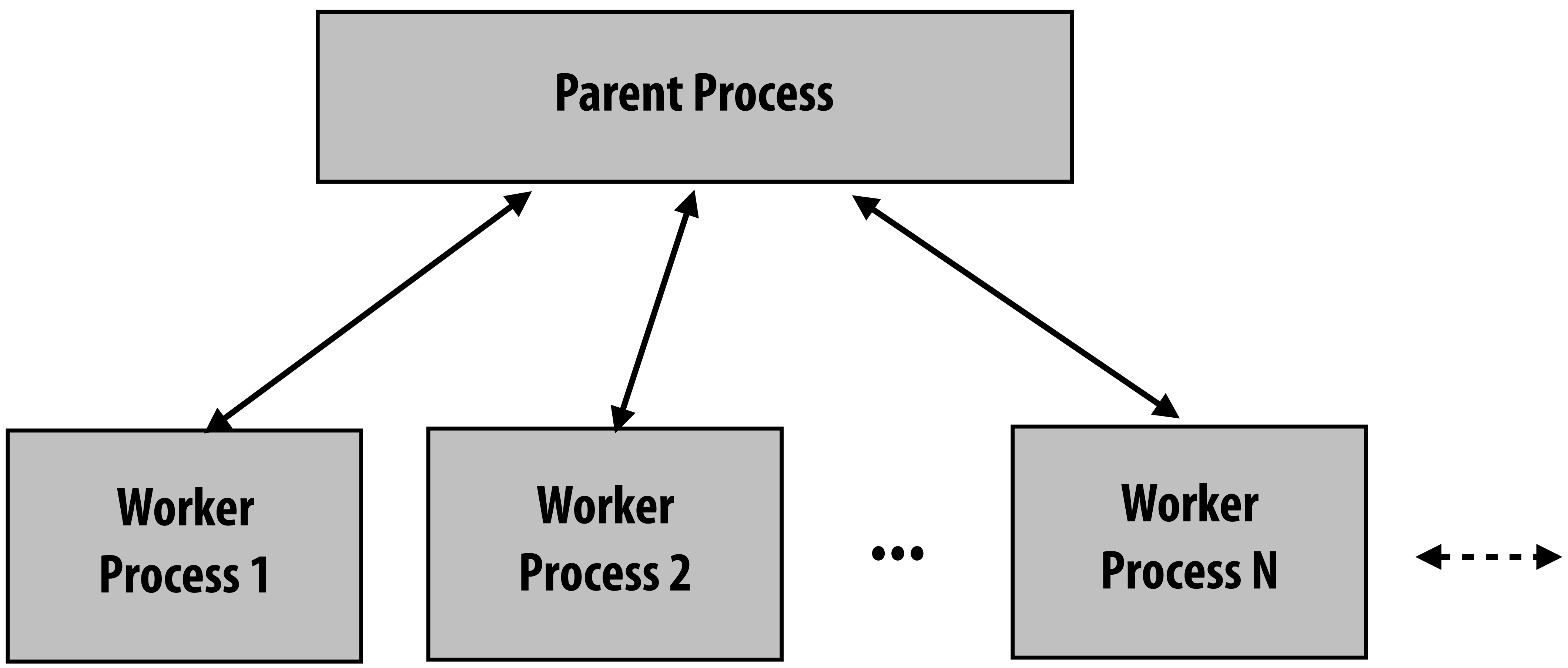
- Many other issues in developing a successful web platform
  - Reliability, security, privacy, etc.
  - There are other great courses at CMU for this

A simple web server for static content

```
while (1)
{

    request = wait_for_request();

    filename = parse_request();

    read file from disk

    send file contents as response

}
```

Is site performance a question of throughput or latency?

A simple parallel web server

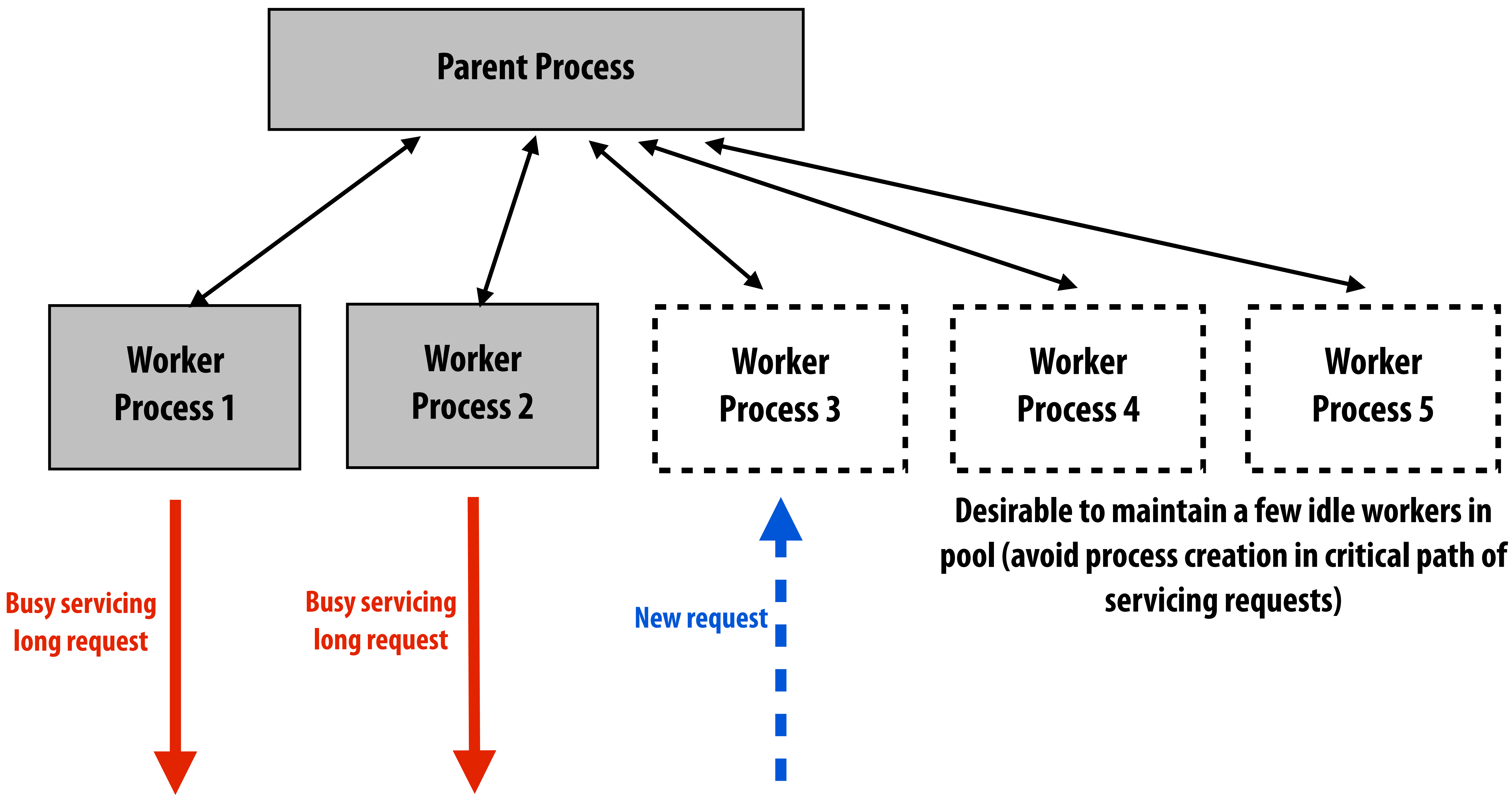



```
while (1)
{
    request = wait_for_request();
    filename = parse_request();
    read file from disk
    send file contents as response
}
```

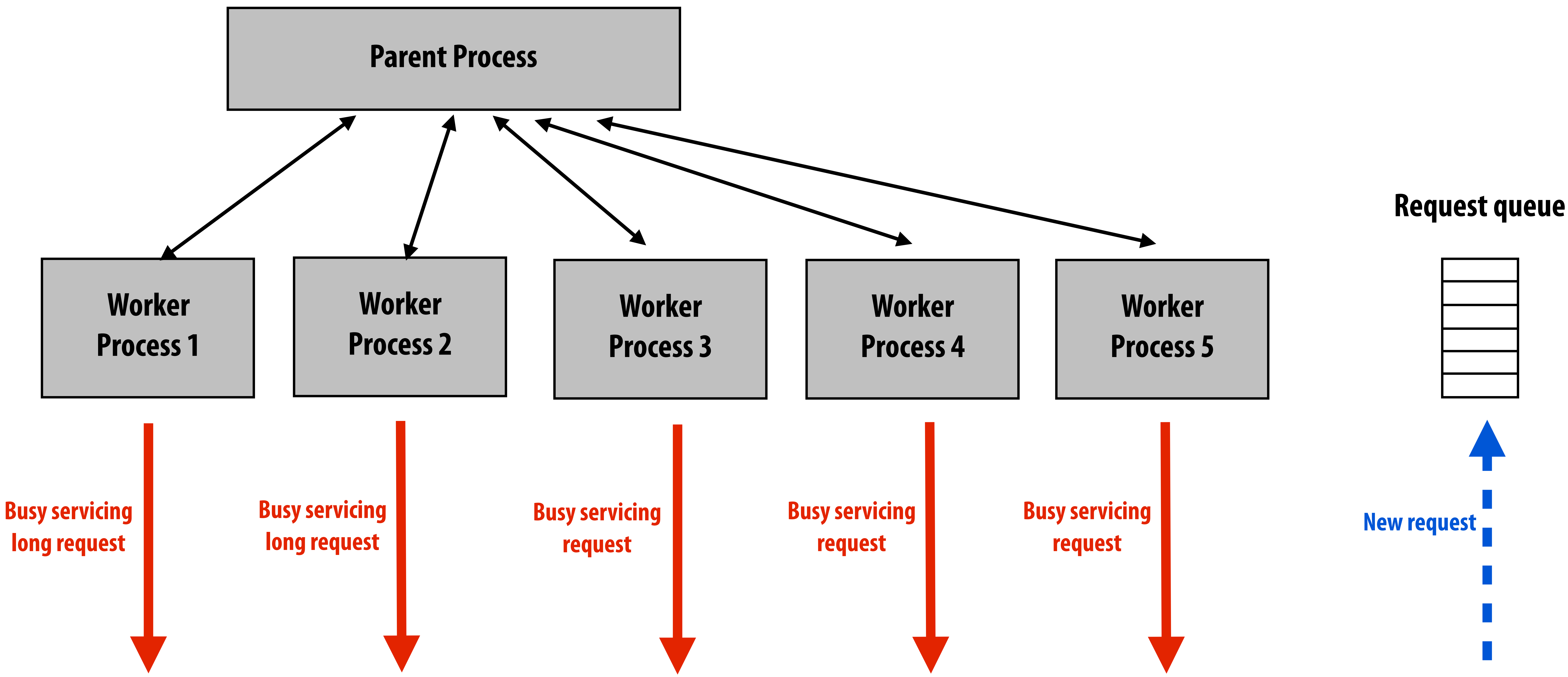
What factors would you consider in setting the value of N?

- Parallelism: use all the server's cores
- Latency hiding: hide long-latency disk read operations (by context switching between worker processes)
- Concurrency: many outstanding requests, want to service quick requests while long requests are in progress (e.g., large file transfer)
- Footprint: don't want too many threads that thrashing occurs

Example: Apache's parent process dynamically manages size of worker pool



Limit maximum number of workers to avoid excessive footprint (memory thrashing)



Key parameter of Apache's "prefork" multi-processing module: `MaxRequestWorkers`

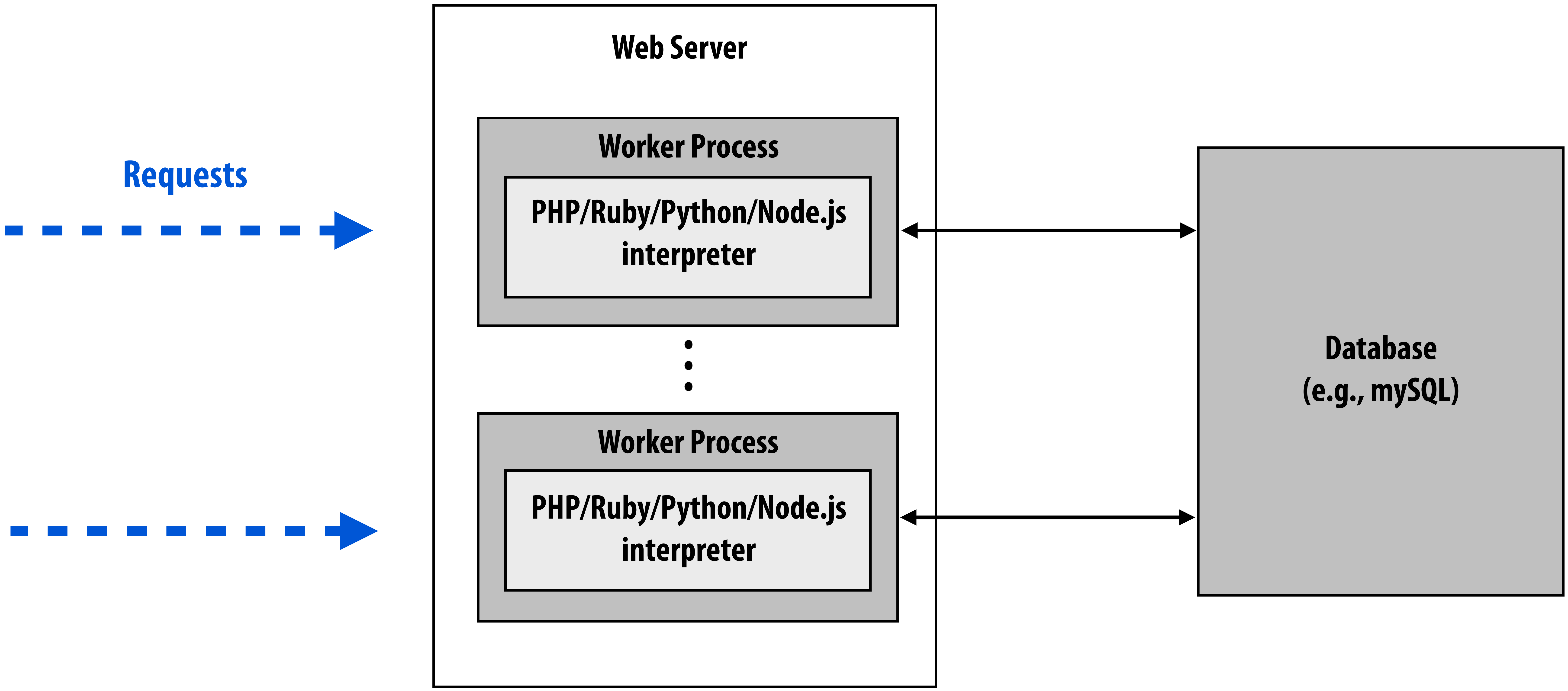
Aside: why partition into processes, not threads?

- **Protection**
  - Don't want a crash of one worker to bring down the whole web server
  - Often want to use non-thread safe libraries (e.g., third-party libraries) in server operation

- **Parent process can periodically recycle workers**
  (robustness to memory leaks)

- **Of course, multi-threaded web server solutions exist as well**
  (e.g., Apache's "worker" module)

Dynamic web content



"Response" is not a static page on disk, but the result of web-application logic running in response to a request.

Consider the amount of logic and the number database queries required to generate your Facebook News Feed.

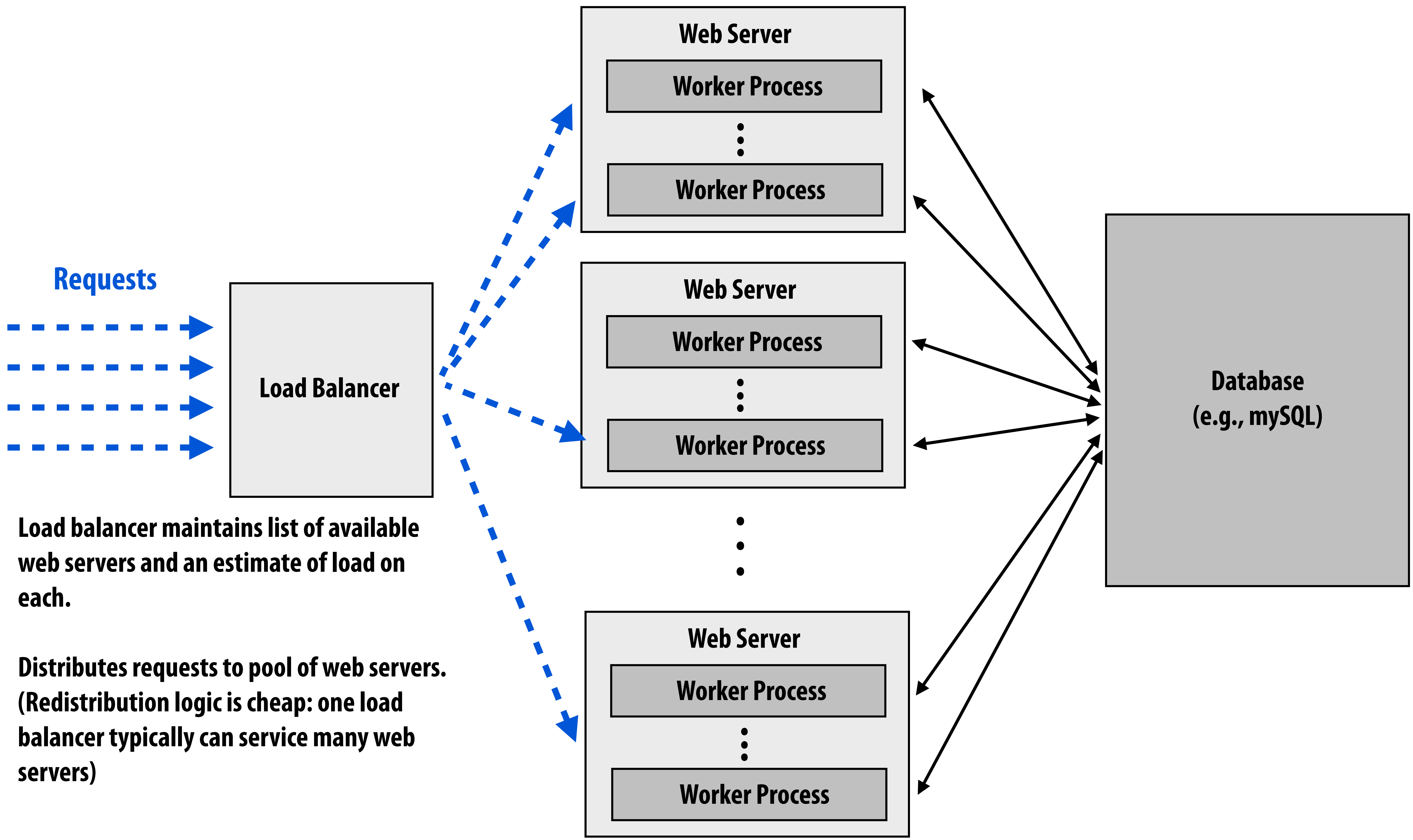
Basic performance (poor)

- Two popular content management systems (PHP)
  - Wordpress ~ 12 requests/sec/core (DB size = 1000 posts)
  - MediaWiki ~ 8 requests/sec/core

[Source: Talaria Inc.]

"Scale out" to increase throughput

Use many web servers to meet site throughput goals.

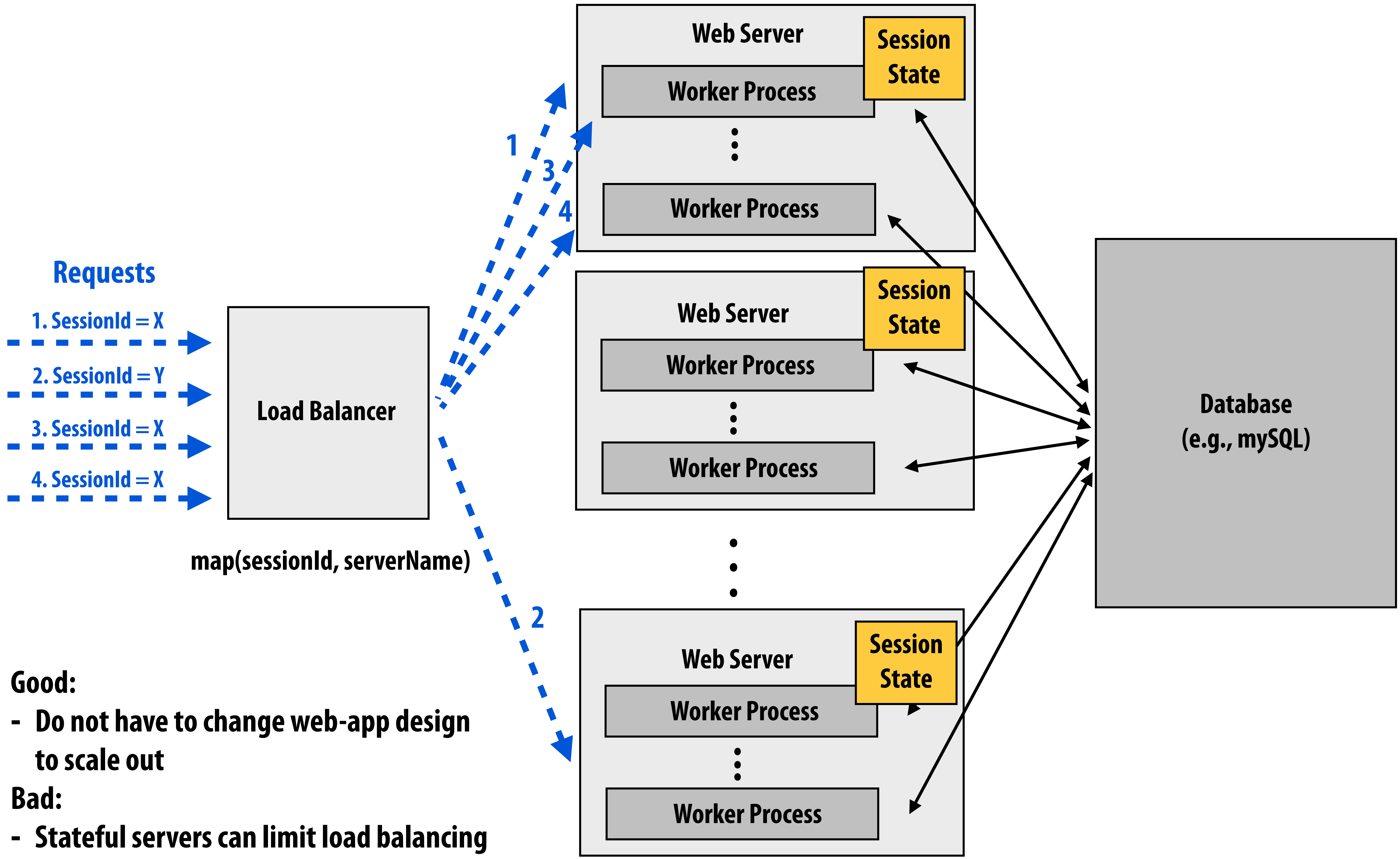


Load balancer maintains list of available web servers and an estimate of load on each.

Distributes requests to pool of web servers. (Redistribution logic is cheap: one load balancer typically can service many web servers)

Load balancing with persistence

All requests associated with a session are directed to the same server (aka. session affinity, "sticky sessions")

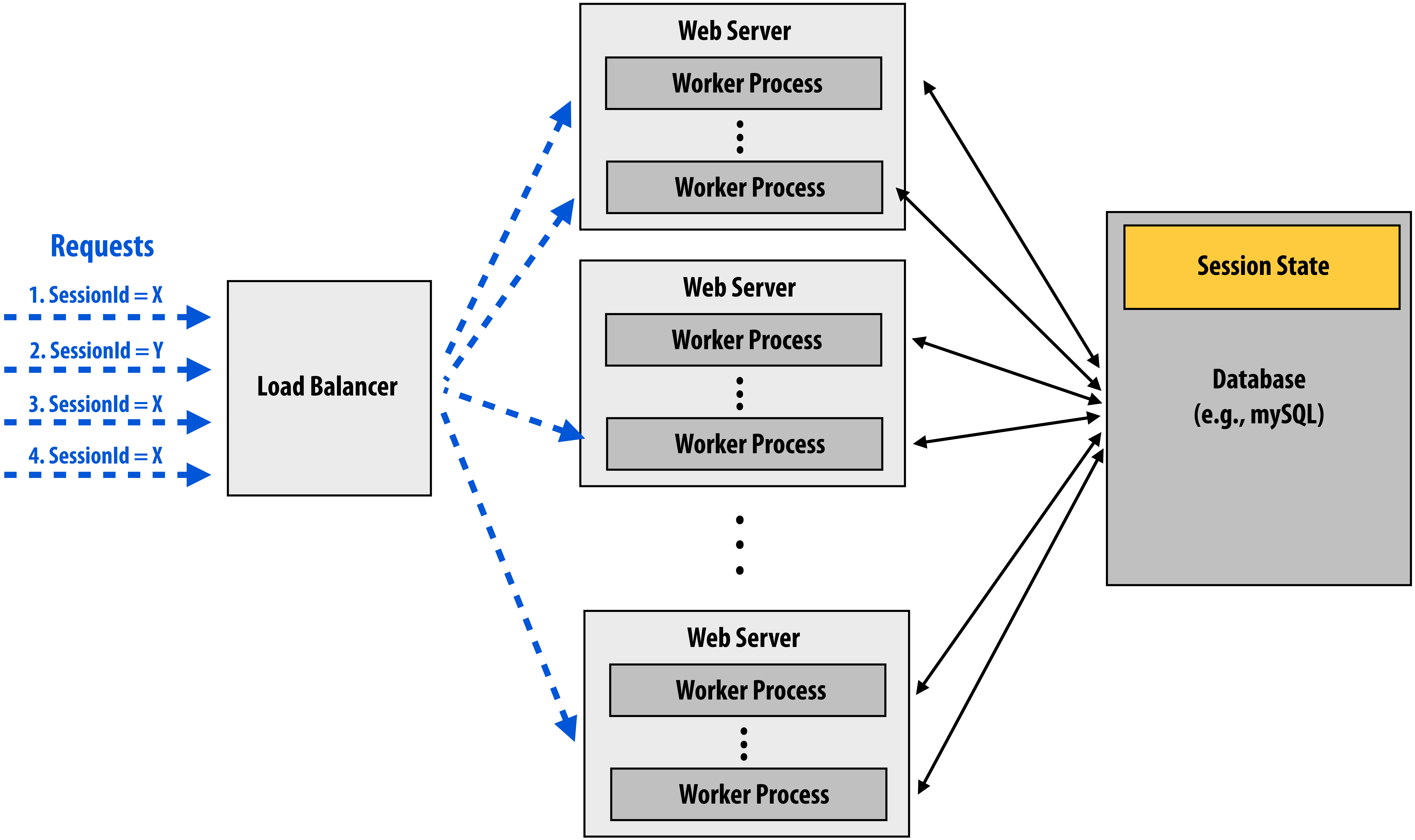


Good:
- Do not have to change web-app design to scale out

Bad:
- Stateful servers can limit load balancing ability. Also, session is lost if server fails

Desirable: avoid persistent state in web server

Maintain stateless servers, treat sessions as persistent data to be stored in the DB.

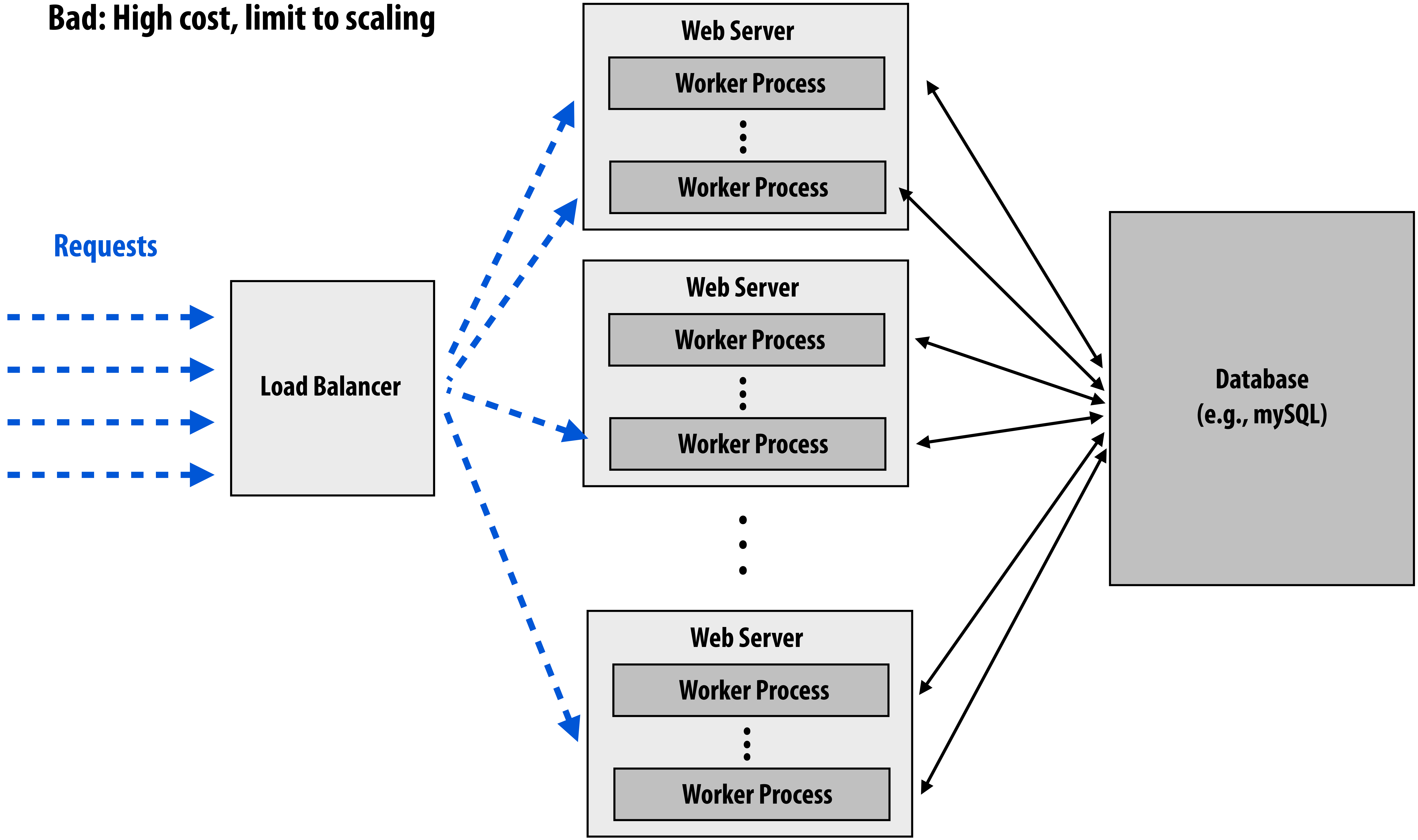


Dealing with database contention

Option 1: "scale up": buy better hardware for database server, buy professional-grade DB that scales

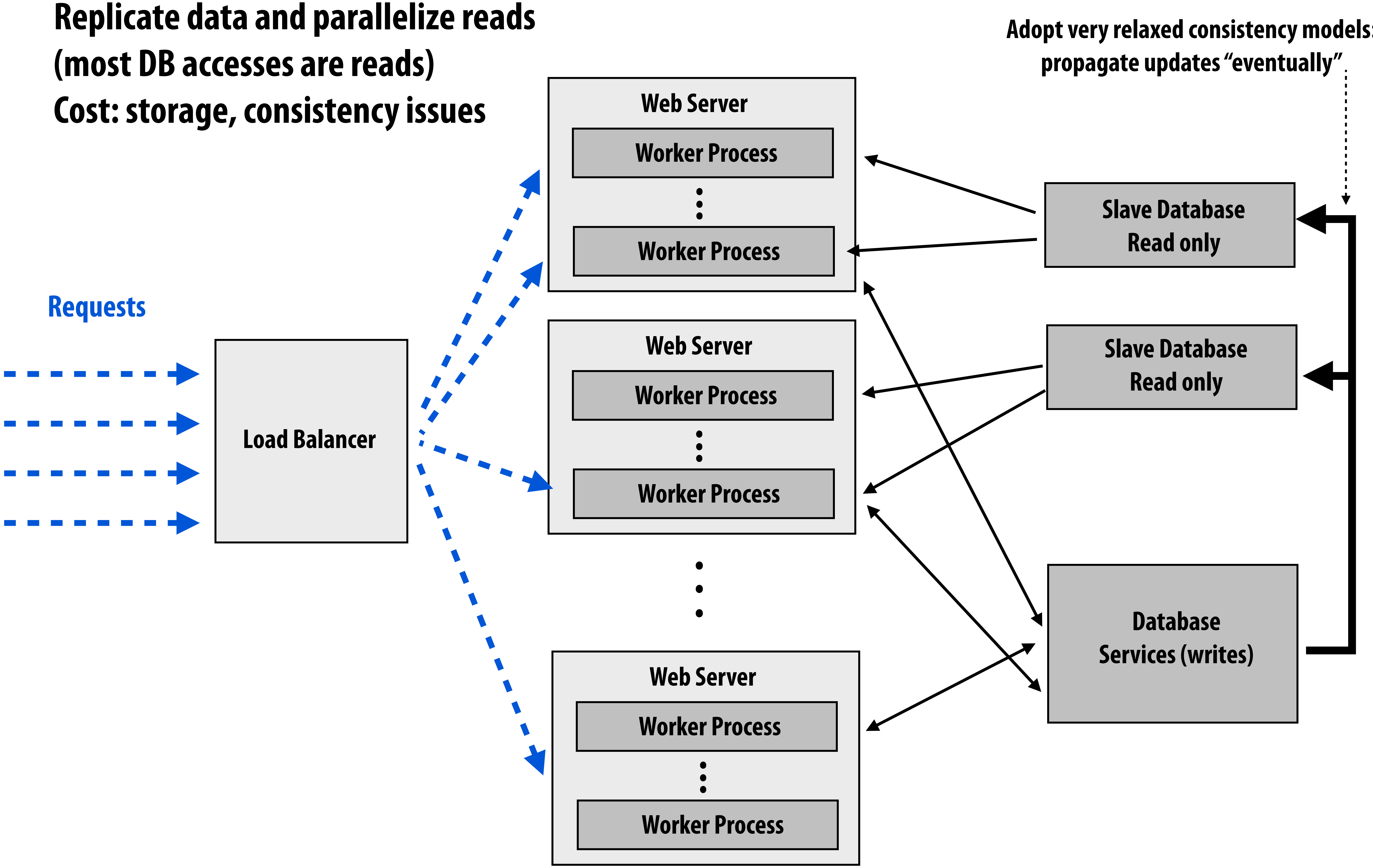
Good: no change to software

Bad: High cost, limit to scaling



Scaling out a database: replicate

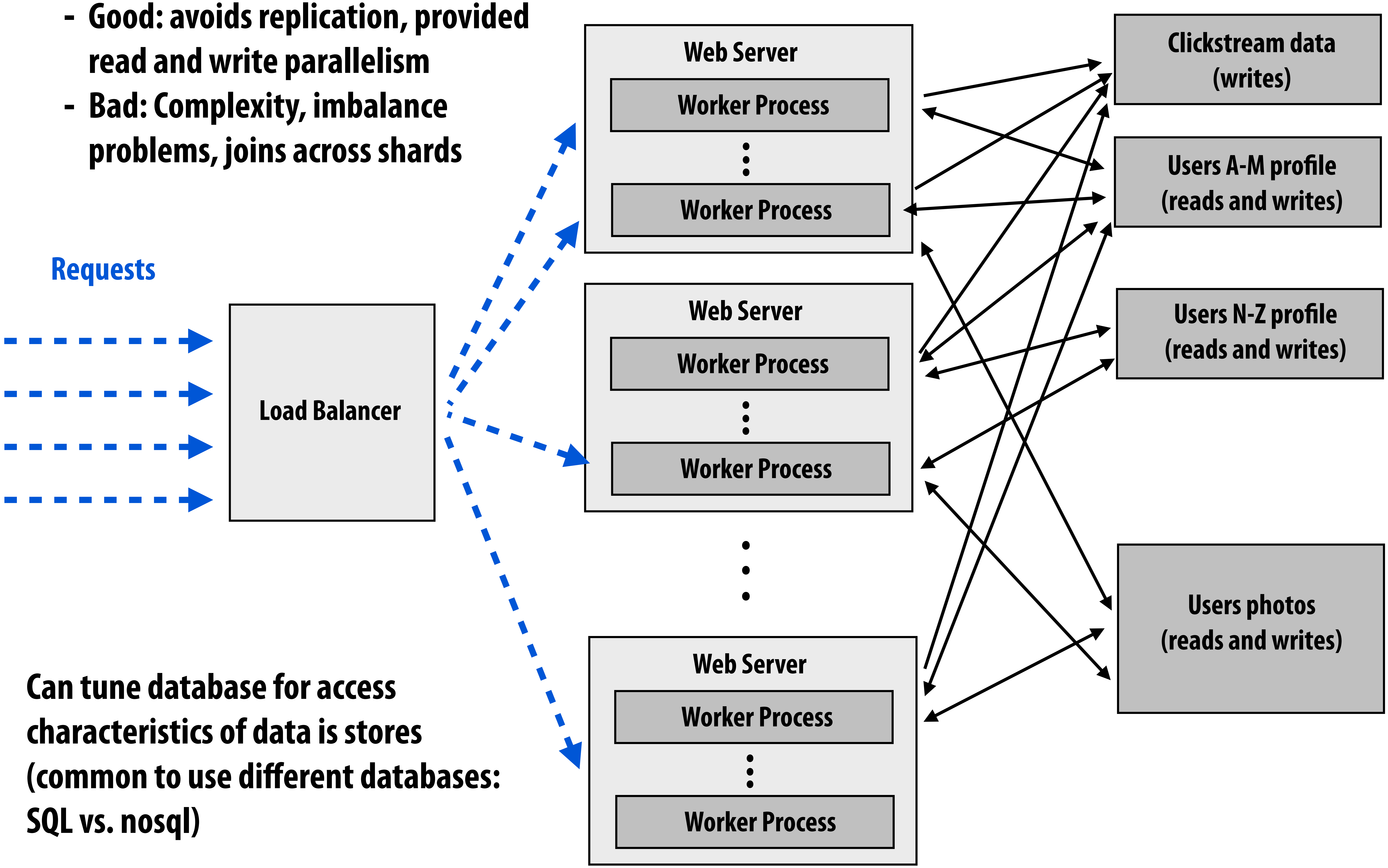
Replicate data and parallelize reads
(most DB accesses are reads)
Cost: storage, consistency issues



Scaling out a database: partition

Partition data ("shard")

- Good: avoids replication, provided read and write parallelism
- Bad: Complexity, imbalance problems, joins across shards



Can tune database for access characteristics of data is stores (common to use different databases: SQL vs. nosql)

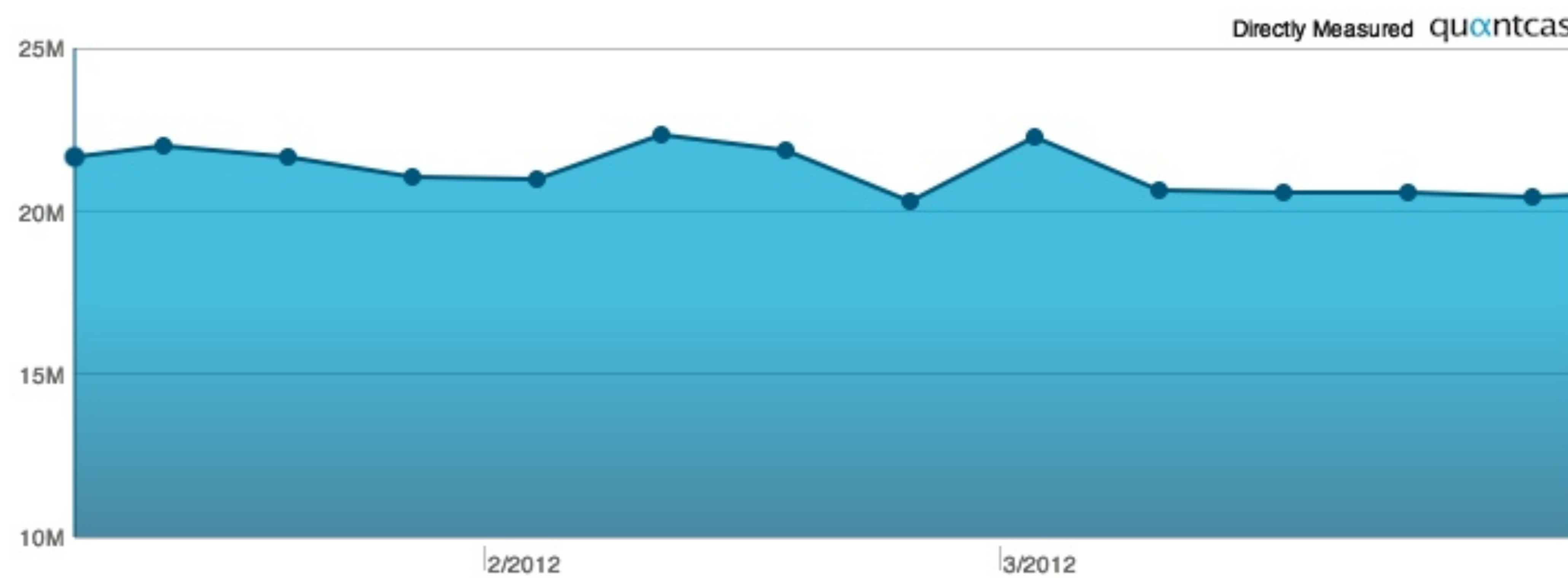
How many web servers do you need?

Web traffic is bursty

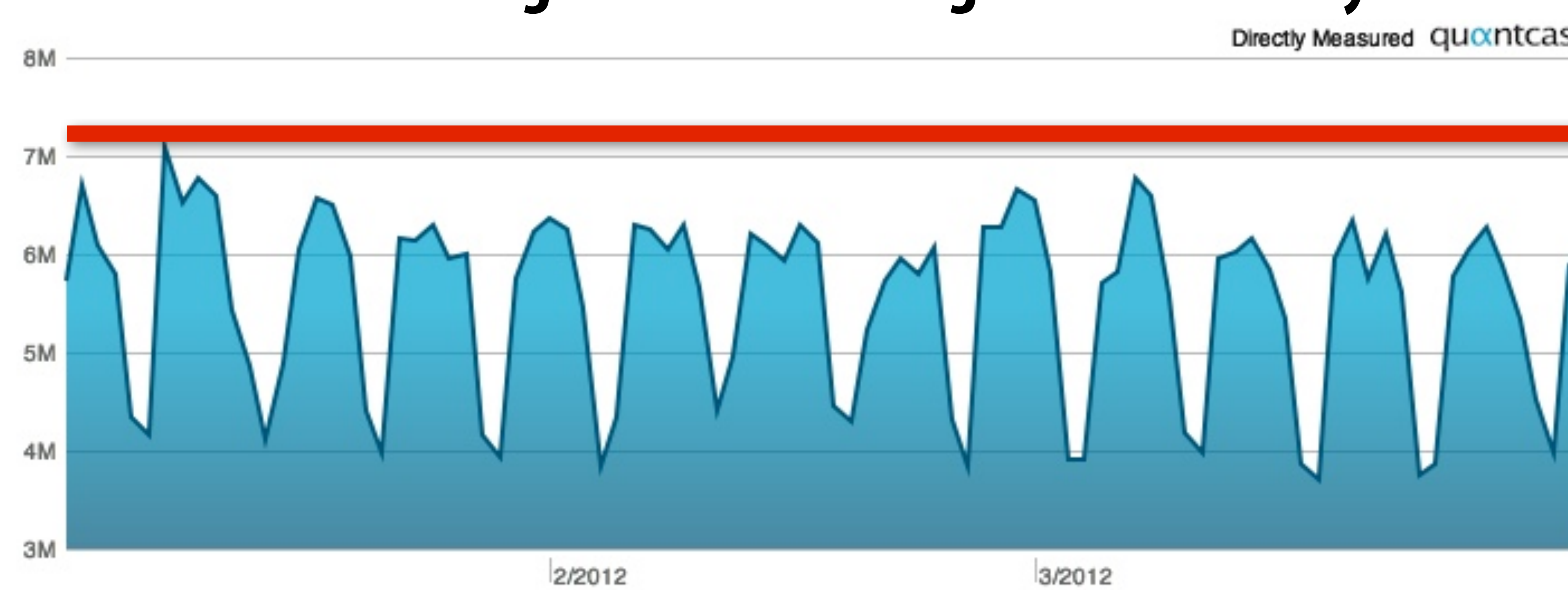
Amazon.com Page Views

Holiday shopping season

HuffingtonPost.com Page Views Per Week



HuffingtonPost.com Page Views Per Day



(less people read news on weekends)

More examples:

- Facebook gears up for bursts of image uploads on Halloween and New Year's Eve.
- Twitter topics trend after world events

Problem

- Site load is bursty

- Provisioning site for the average case load will result in poor quality of service (or failures) during peak usage
  - Peak usage tends to be when users care the most... since by the definition the site is important at these times

- Provisioning site for the peak usage case will result in many idle servers most of the time
  - Not cost efficient

Elasticity!

- Main idea: site automatically adds or shuts down web servers based on measured load

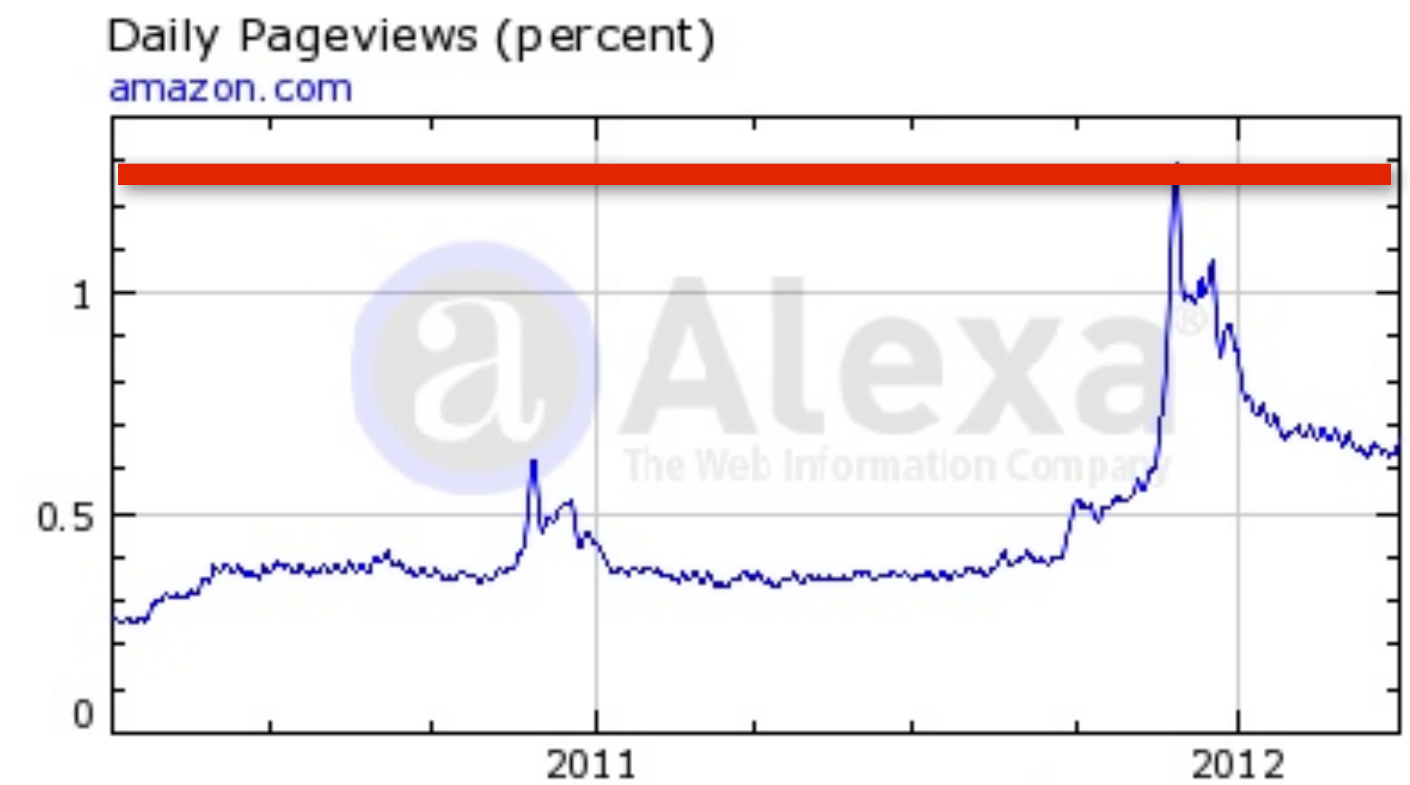
- Need source of servers available on-demand
  - Example: Amazon.com EC2 instances

Example: Amazon's elastic compute cloud (EC2)

- Amazon had an over-provisioning problem

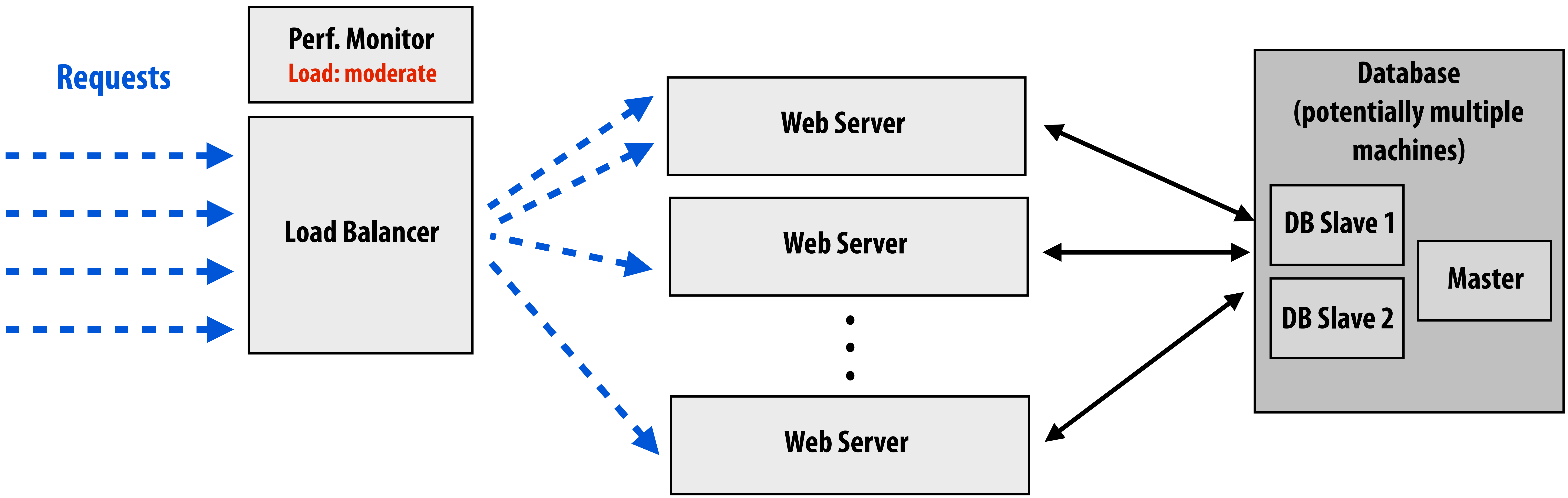
- Solution: make machines available for rent to others in need of compute
  - For those that don't want to incur cost of, or have expertise to, manage own machines at scale
  - For those that need elastic compute capability

Amazon.com Page Views



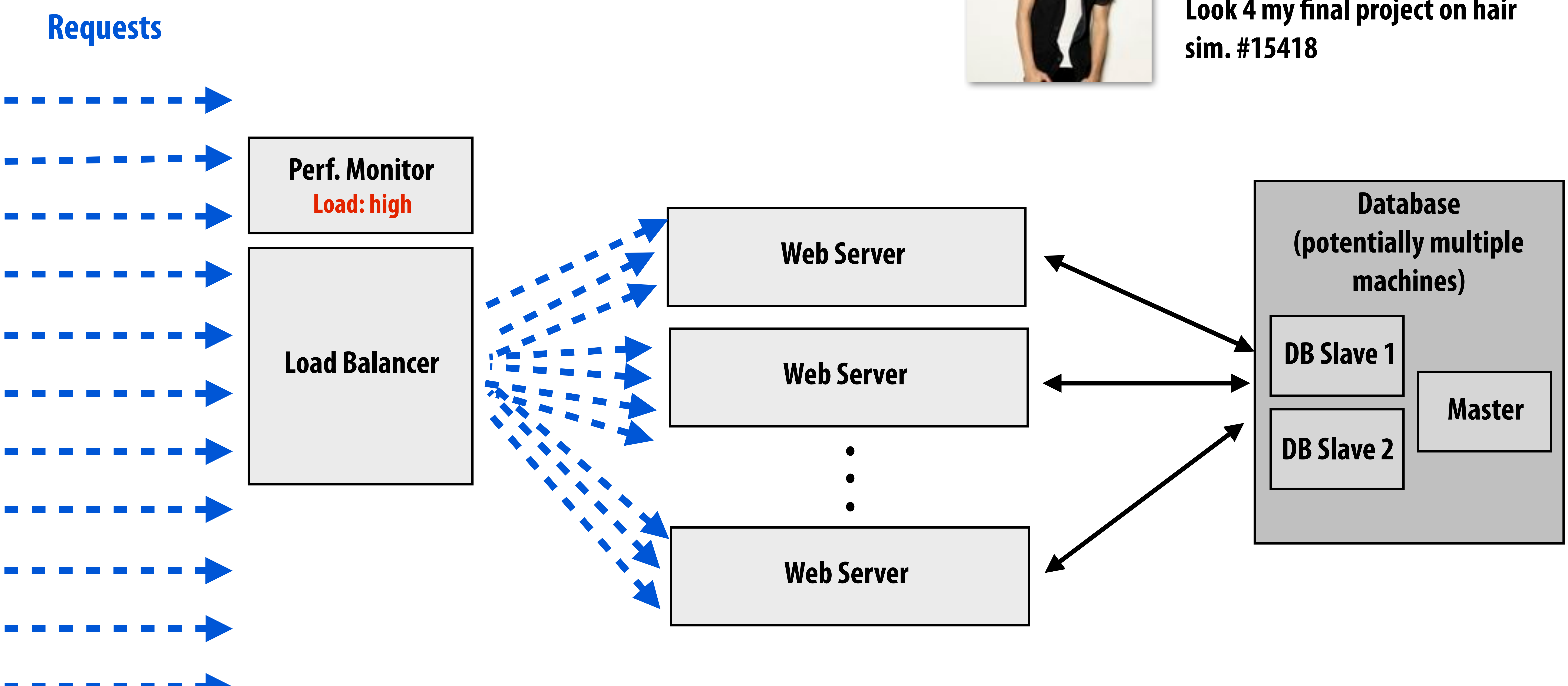
	Linux/UNIX Usage	Windows Usage
Standard On-Demand Instances		
Small (Default)	$0.080 per Hour	$0.115 per Hour
Medium	$0.160 per Hour	$0.230 per Hour
Large	$0.320 per Hour	$0.460 per Hour
Extra Large	$0.640 per Hour	$0.920 per Hour
Micro On-Demand Instances		
Micro	$0.020 per Hour	$0.030 per Hour
Hi-Memory On-Demand Instances		
Extra Large	$0.450 per Hour	$0.570 per Hour
Double Extra Large	$0.900 per Hour	$1.140 per Hour
Quadruple Extra Large	$1.800 per Hour	$2.280 per Hour
Hi-CPU On-Demand Instances		
Medium	$0.165 per Hour	$0.285 per Hour
Extra Large	$0.660 per Hour	$1.140 per Hour
Cluster Compute Instances		
Quadruple Extra Large	$1.300 per Hour	$1.610 per Hour
Eight Extra Large	$2.400 per Hour	$2.970 per Hour
Cluster GPU Instances		
Quadruple Extra Large	$2.100 per Hour	$2.600 per Hour

Site configuration: normal load



Event triggers spike in load

@justinbieber: OMG, parallel prog. class at CMU is awesome. Look 4 my final project on hair sim. #15418



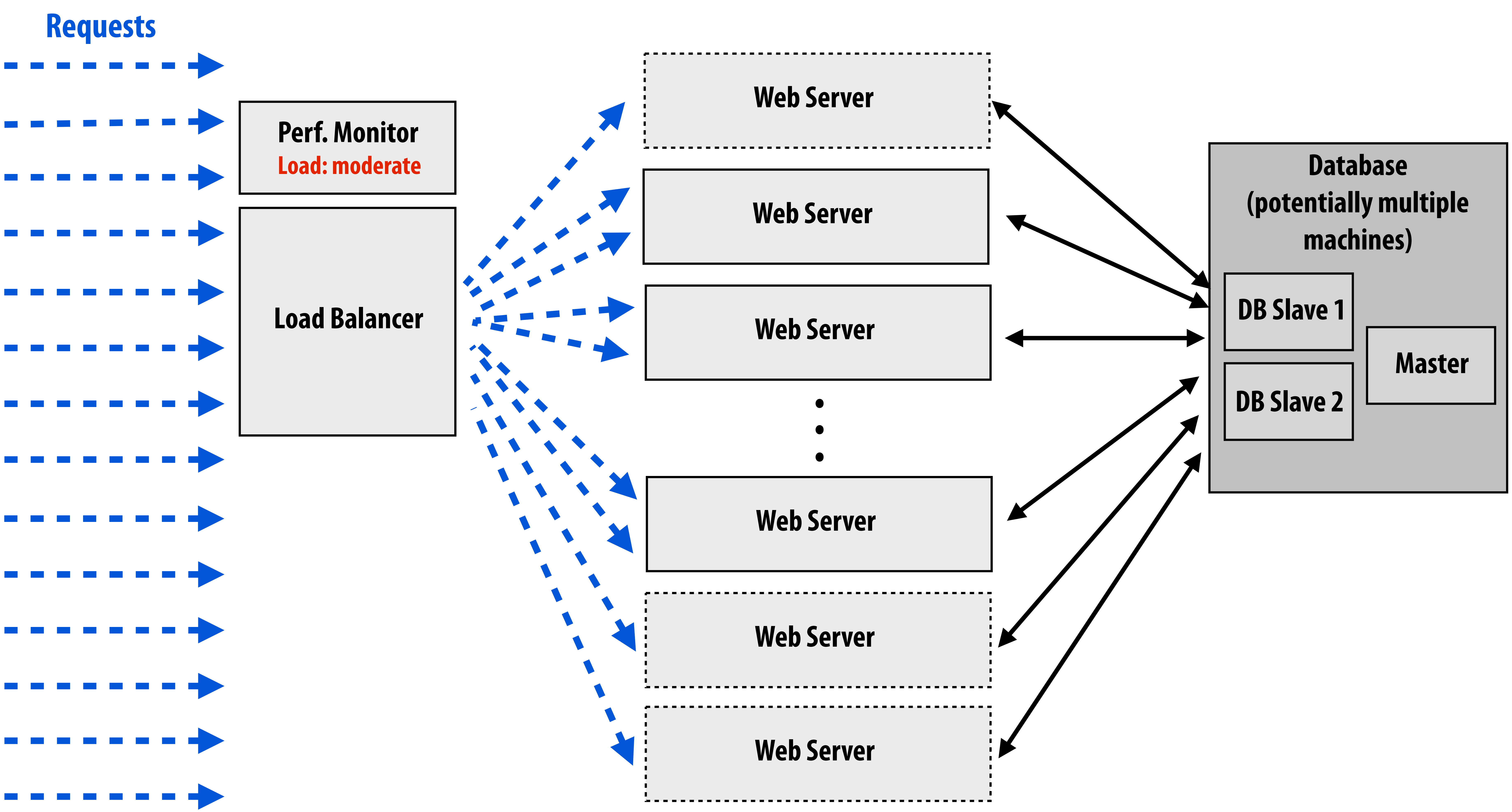
Heavily loaded servers: slow response times

Site configuration: high load

Site performance monitor detects high load

Instantiates new web server instances

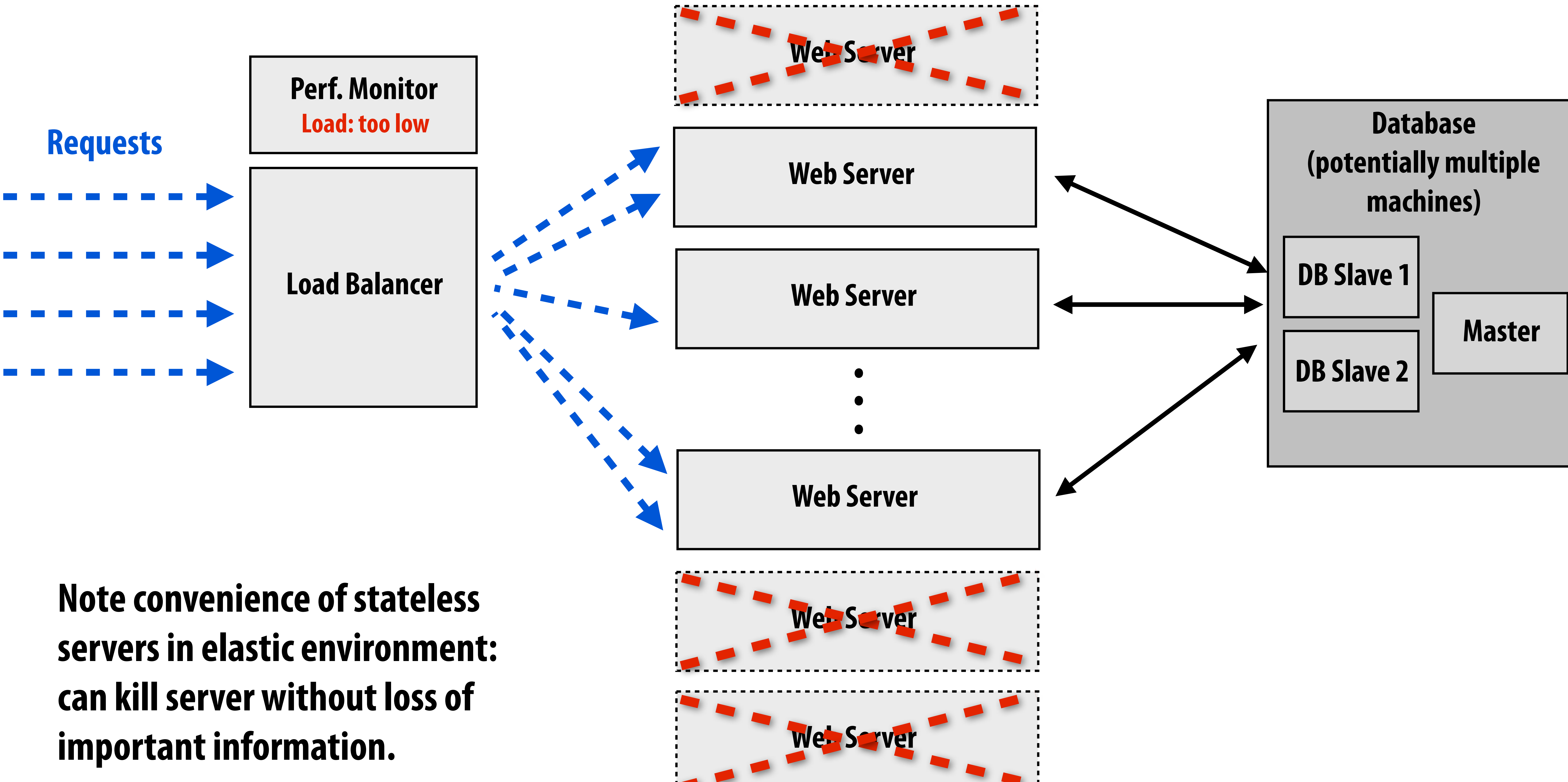
Informs load balancer about presence of new servers



Site configuration: return to normal load

Site performance monitor detects low load
Kills extra server instances (to save operating cost)
Informs load balancer about loss of servers

@justinbieber: WTF, parallel programming is 2 hrd. Buy my new album.



Note convenience of stateless servers in elastic environment: can kill server without loss of important information.

Today: many “turn-key” environment-in-a-box services

Offer elastic computing environments for web applications

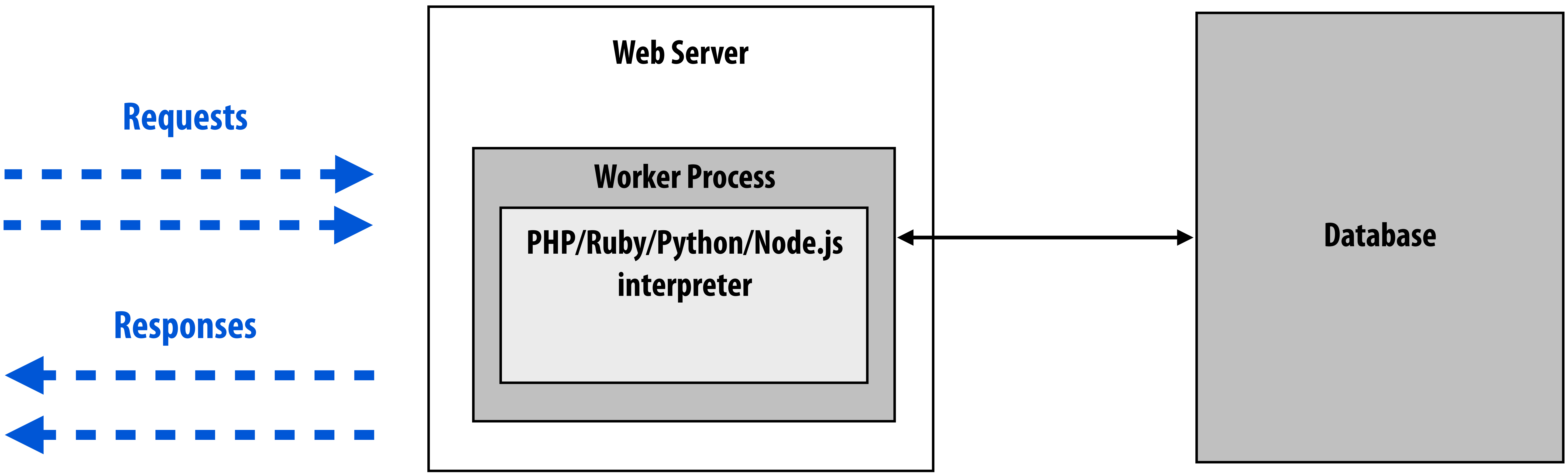
CloudWatch+Auto Scaling
Amazon Elastic Beanstalk

The story so far: parallelism
scale out, scale out, scale out

(+ elasticity to be able to scale out on demand)

Now: reuse and locality

Recall: basic site configuration



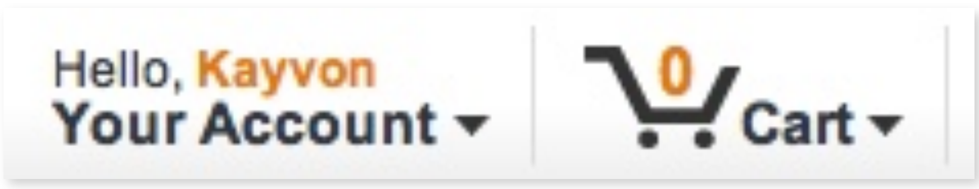
Example PHP Code

```
$query = "SELECT * FROM users WHERE username=‘kayvonf’;
$user = mysql_fetch_array(mysql_query($userquery));

echo “<div>” . $user[‘FirstName’] . “ “ . $user[‘LastName’] . “</div>”;
```

Response Information Flow

`<div>Kayvon Fatahalian</div>`

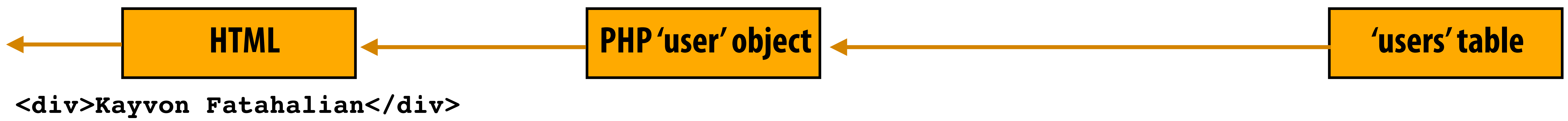
Work repeated every page



Example PHP Code

```
$query = "SELECT * FROM users WHERE username='kayvonf';
$user = mysql_fetch_array(mysql_query($userquery));

echo "<div>" . $user['FirstName'] . " " . $user['LastName'] . "</div>";
```

Response Information Flow

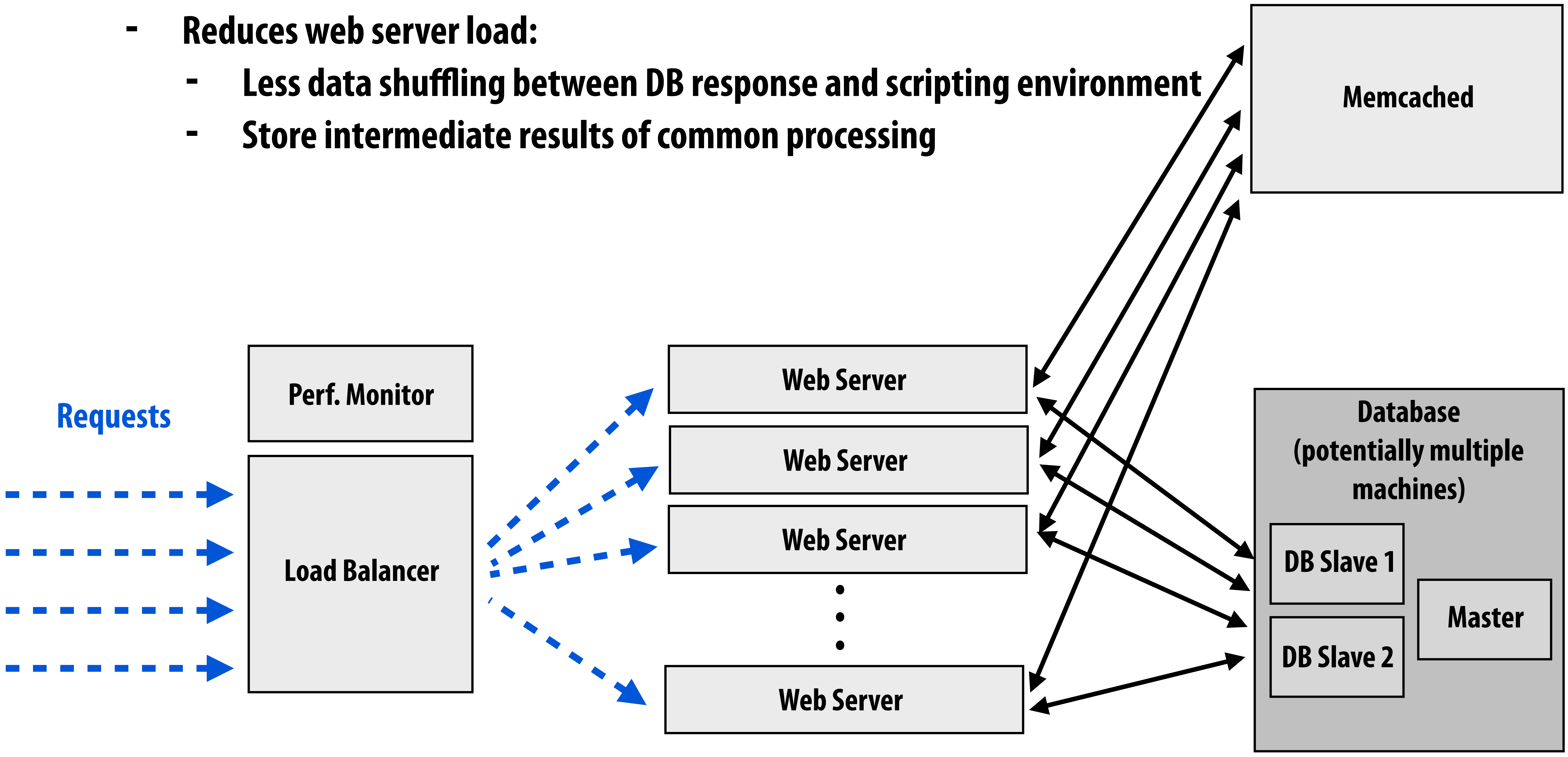


- Steps repeated to emit my name at the top of every page:
  - Communicate with DB
  - Perform query
  - Marshall results from database into object model of scripting language
  - Generate presentation
  - etc...

Remember, DB can be hard to scale!

Solution: cache!

- Cache commonly accessed objects
  - Example: memcached, in memory key-value store (e.g, a big hash table)
  - Reduces database load (fewer queries)
  - Reduces web server load:
    - Less data shuffling between DB response and scripting environment
    - Store intermediate results of common processing

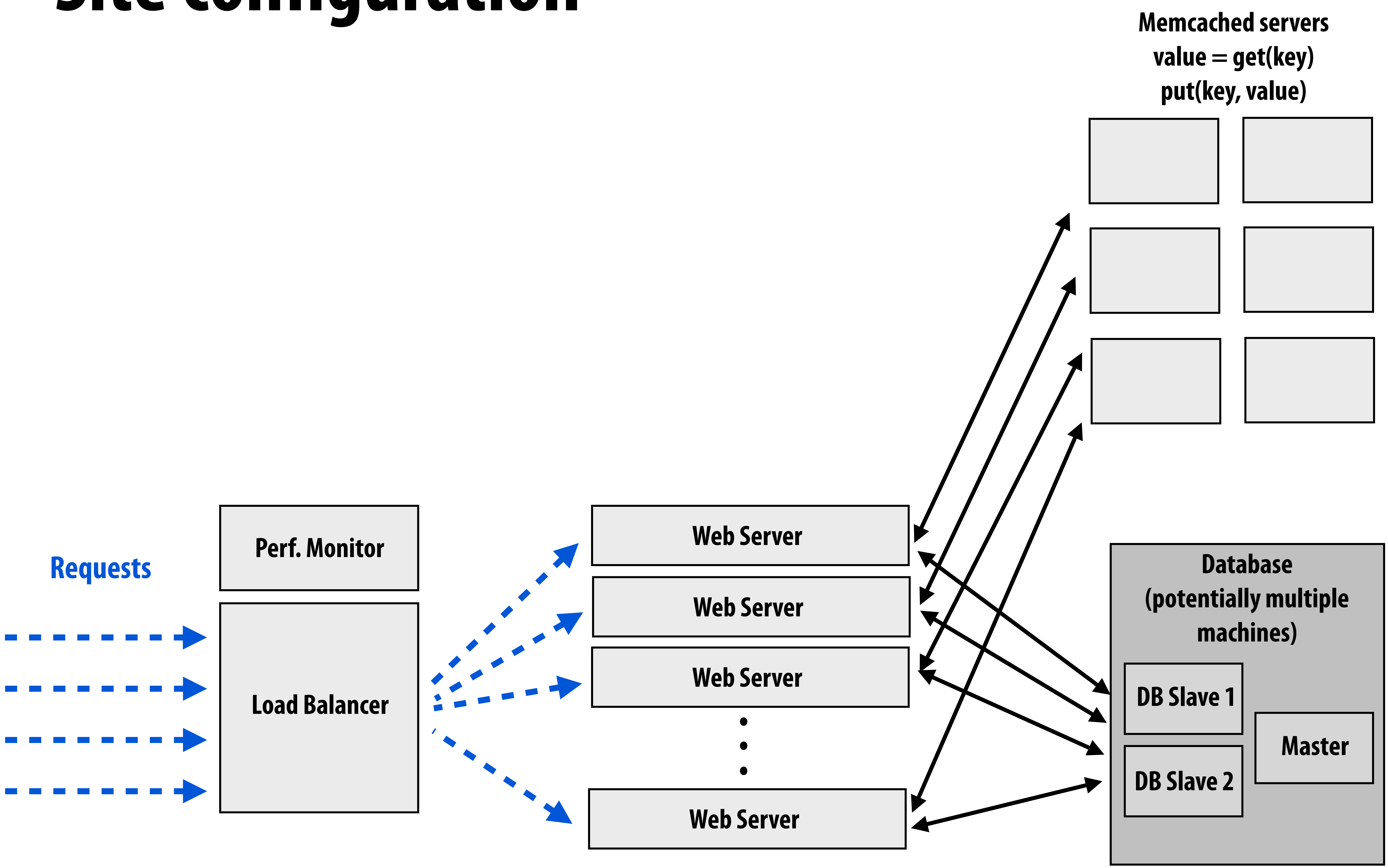


Caching example

```
userid = $_SESSION['userid'];

check if memcache->get(userid) retrevies a valid user object

if not:
   make expensive database query
   add resulting object into cache with memcache->put()
   (so future requests involving this user can skip the query)

continue with request processing logic
```

- Obviously, there is complexity associated with keeping caches in sync with data in the DB in the presence of writes
  - Must invalidate cache
  - Simple solution: only cache read-only objects

Site configuration



Example: Facebook memcached deployment

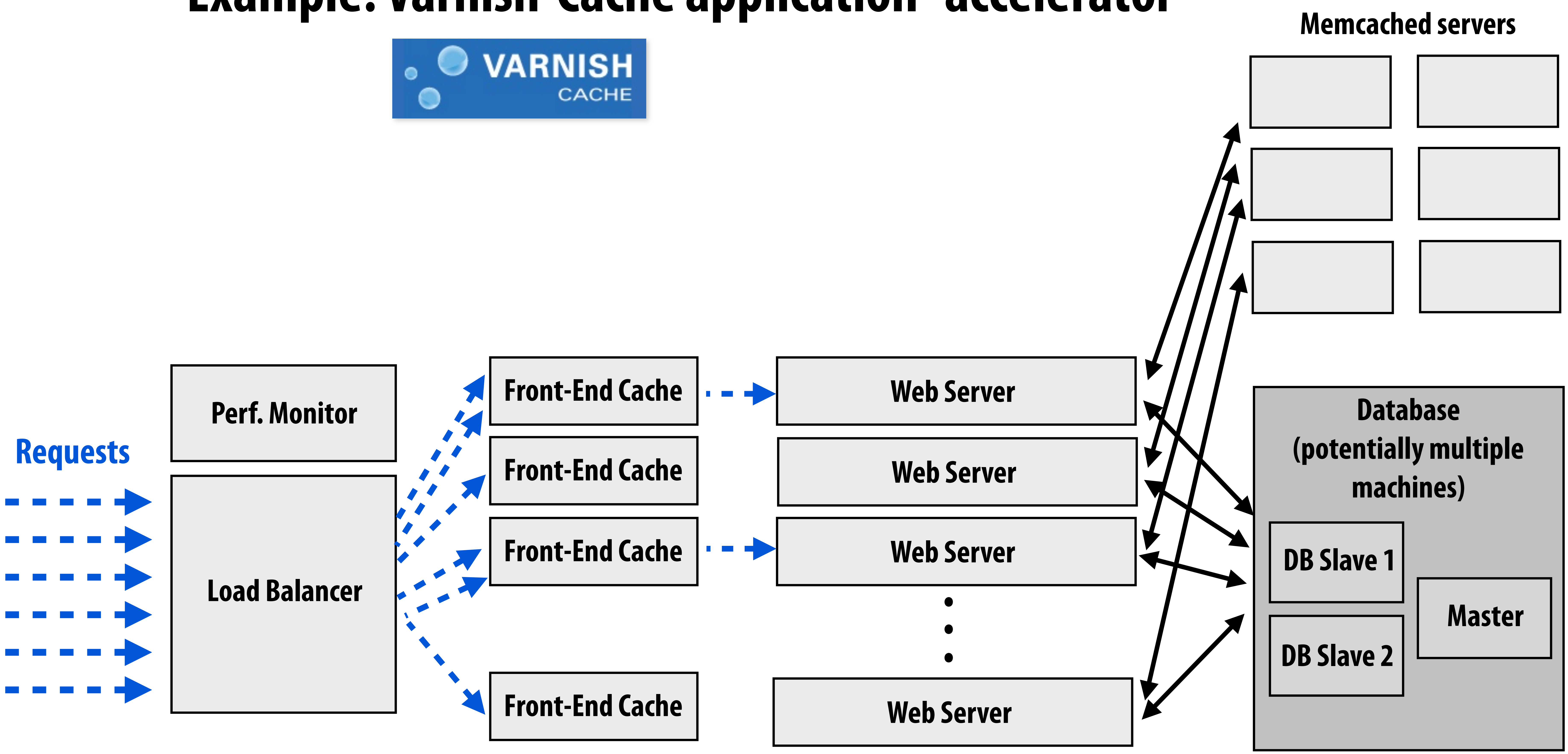
- Facebook, circa 2008
  - 800 memcached servers
  - 28 TB of cached data

- Performance
  - 200,000 UDP requests per second @ 173 msec latency
  - 300,000 UDP requests per second possible at "unacceptable" latency

Source: https://www.facebook.com/note.php?note_id=39391378919

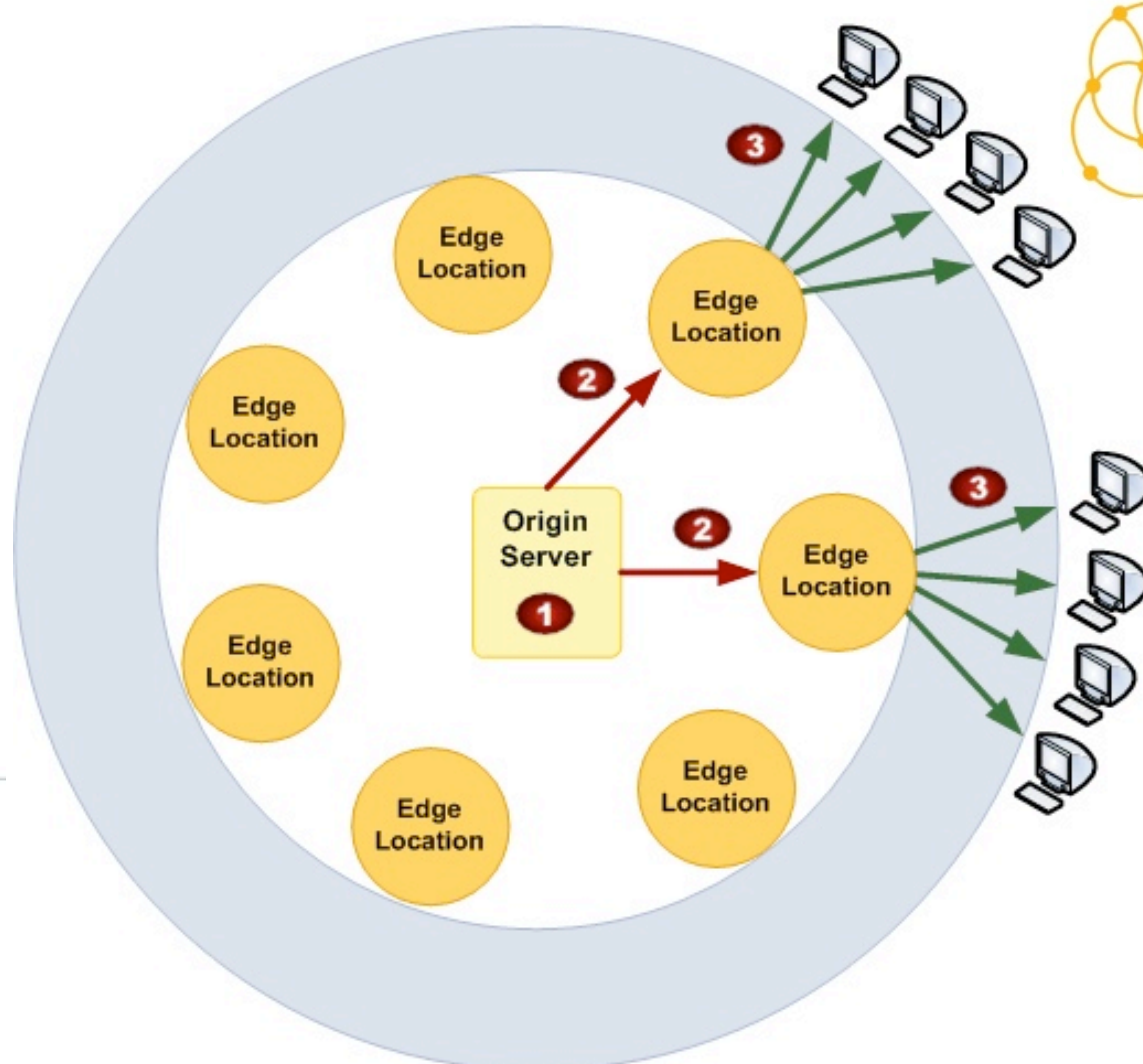
More caching

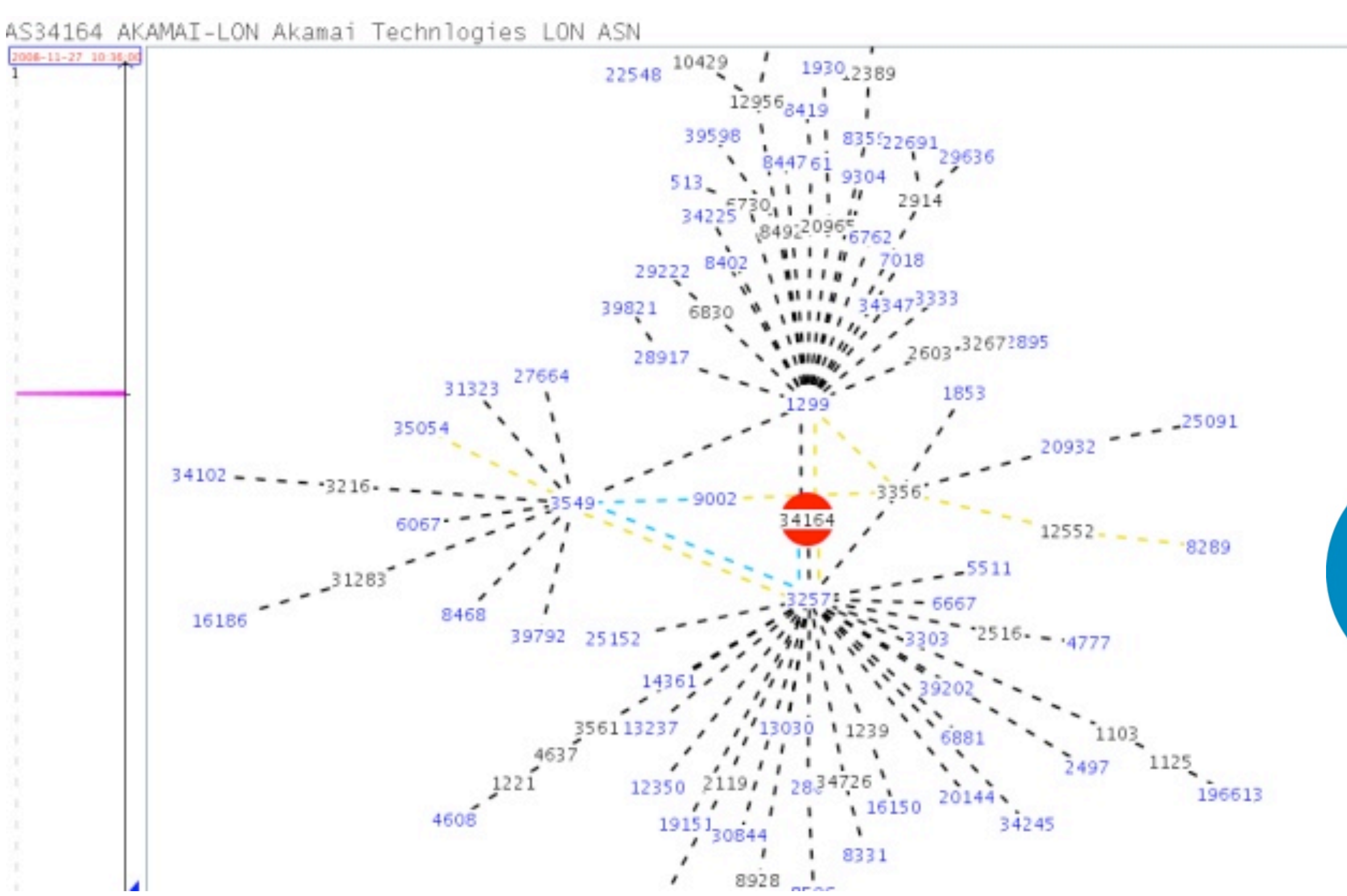
- Cache web server responses (e.g. entire pages, pieces of pages)
  - Reduce load on web servers
  - Example: Varnish-Cache application "accelerator"



Caching using content distribution networks (CDNs)

- Serving large media assets can be expensive to serve (high bandwidth costs, tie up web servers)
  - E.g., images, streaming video
- Physical locality is important
  - Higher bandwidth
  - Lower latency





London Content Distribution Network
Source: http://www.telco2.net/blog/2008/11/amazon_cloudfront_yet_more_tra.html

CDN usage example

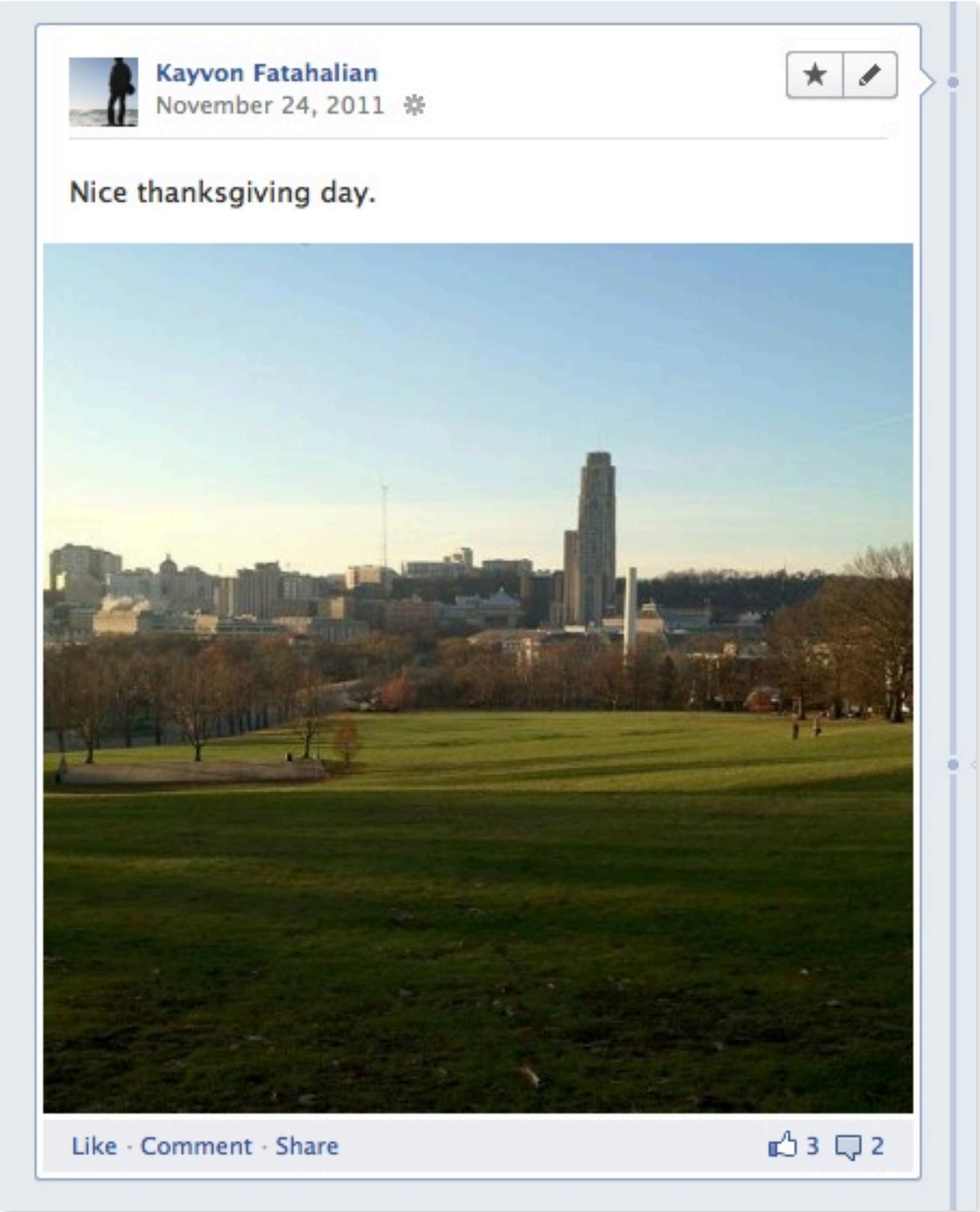
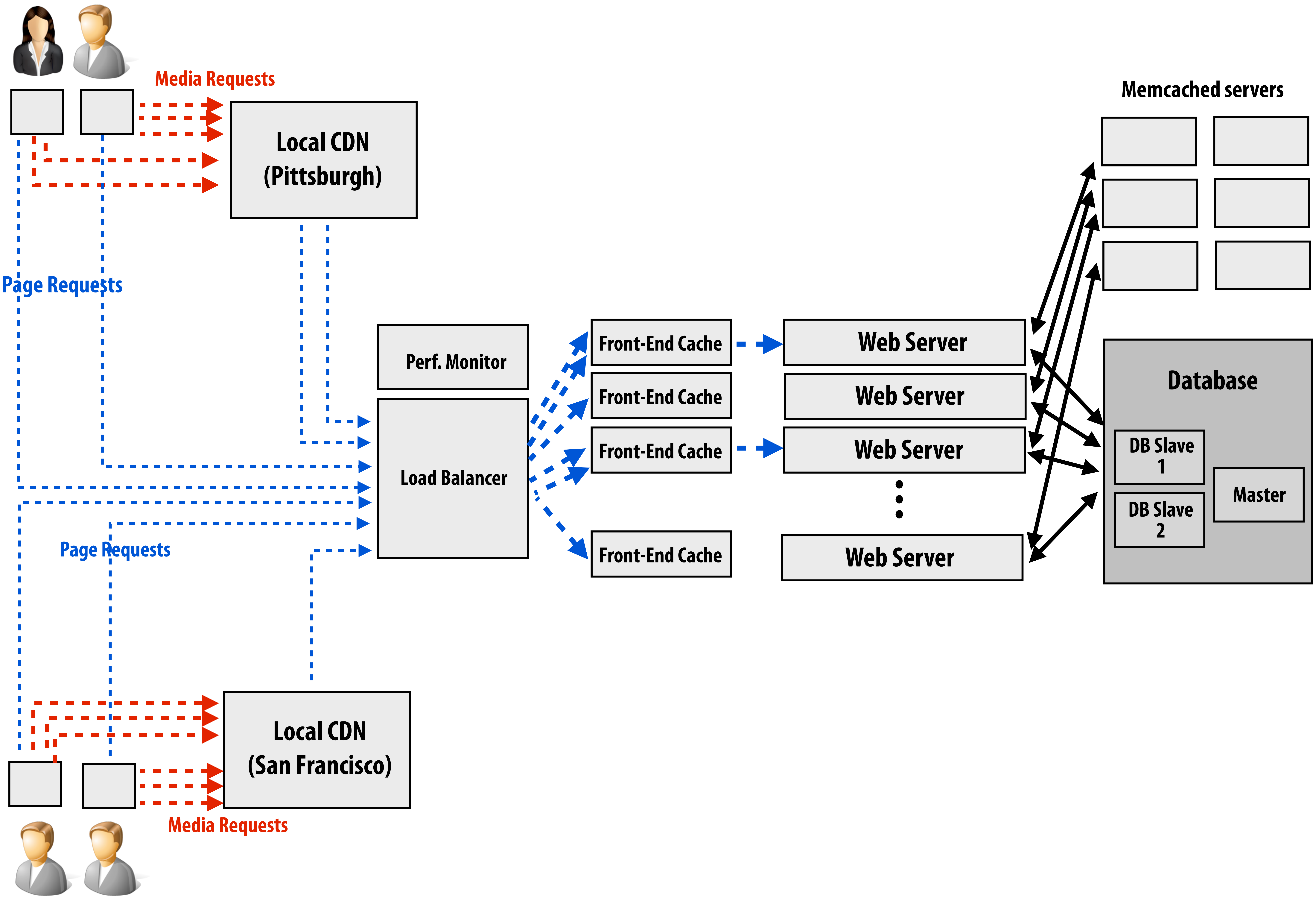


Photo on Facebook Wall:

Page URL:
https://www.facebook.com/photo.php?fbid=10150562736973897&set=a.10150275074093897.338852.722973896

Image source URL:
https://fbcdn-sphotos-a.akamaihd.net/hphotos-ak-ash4/306471_10150401639593897_722973896_8538832_807089003_n.jpg

CDN Integration



Summary: scaling modern web sites

- **Use parallelism**
  - Scale-out parallelism: many web servers
  - Elastic scale-out (cost-effectively adapt to bursty load)
  - Scaling databases can be tricky due to writes (replicate, partition, shard)
- **Exploit locality and reuse**
  - Cache everything (key-value stores)
    - Cache the results of database access (reduce DB load)
    - Cache computation results (reduce web server load)
    - Cache the results of processing requests (reduce web server load)
  - Localize cached data to users, especially for large media content (CDNs)
- **Specialize for performance**
  - Different forms of requests, different workload patterns

Final comment

- **CS student perception:**
  - Web programming is low brow
    (a bunch of entry-level programmers hacking in PHP)
- **Reality:**
  - It is true that:
    - Performance of straight-line <u>application logic</u> is often very poor in these web-programming languages (orders of magnitude left on the table)
    - Programmers writing this code are likely not the world's best coders
  - BUT... <u>scaling</u> a web site is a very challenging parallel-systems problem that involves many of the optimization techniques and design choices studied in this class: just at much larger scales
    - Identifying parallelism and dependencies
    - Workload balancing: static vs. dynamic partitioning issues
    - Data duplication vs. contention
    - Throughput vs. latency trade-offs
    - Parallelism vs. footprint trade-offs
    - Identifying and exploiting reuse and locality