Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 7 – Scalable Request Processing

Learning Objectives

- Describe horizontal scaling approaches
- Explain the strengths and weaknesses of different caching approches
- Explain the WWW caching architecture
- Explain the importance of stateless servers and load balancing

Outline

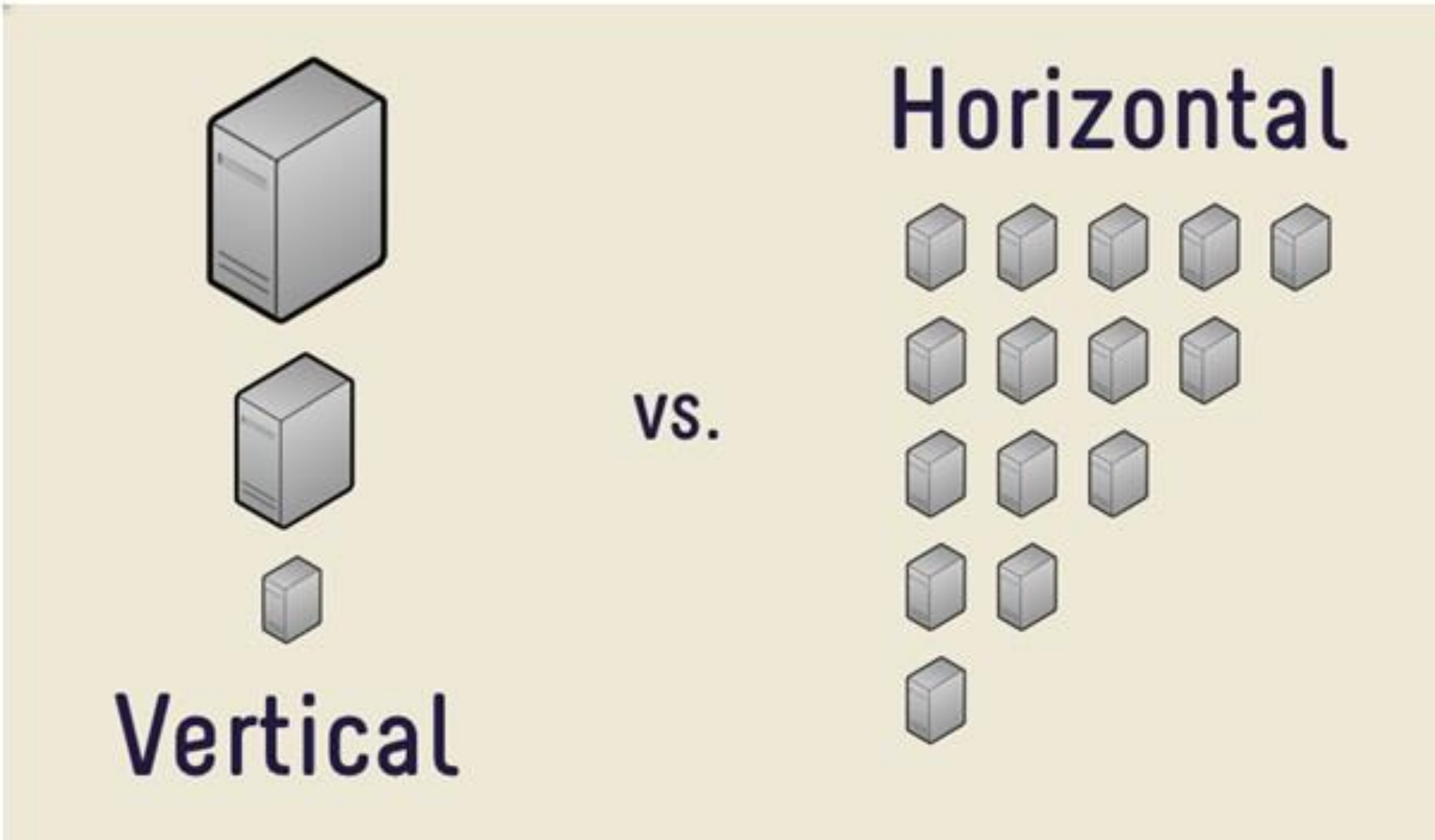
- Scaling the server processing
- Caching approaches
- Web Caching

Scaling the Server Processing

Scalability Basics

- Request processing layer handles client requests
- As volume and frequency of requests grows, we need to scale our processing capacity
- 2 options
  - Scale up
  - Scale out

Scaling Processing Capacity



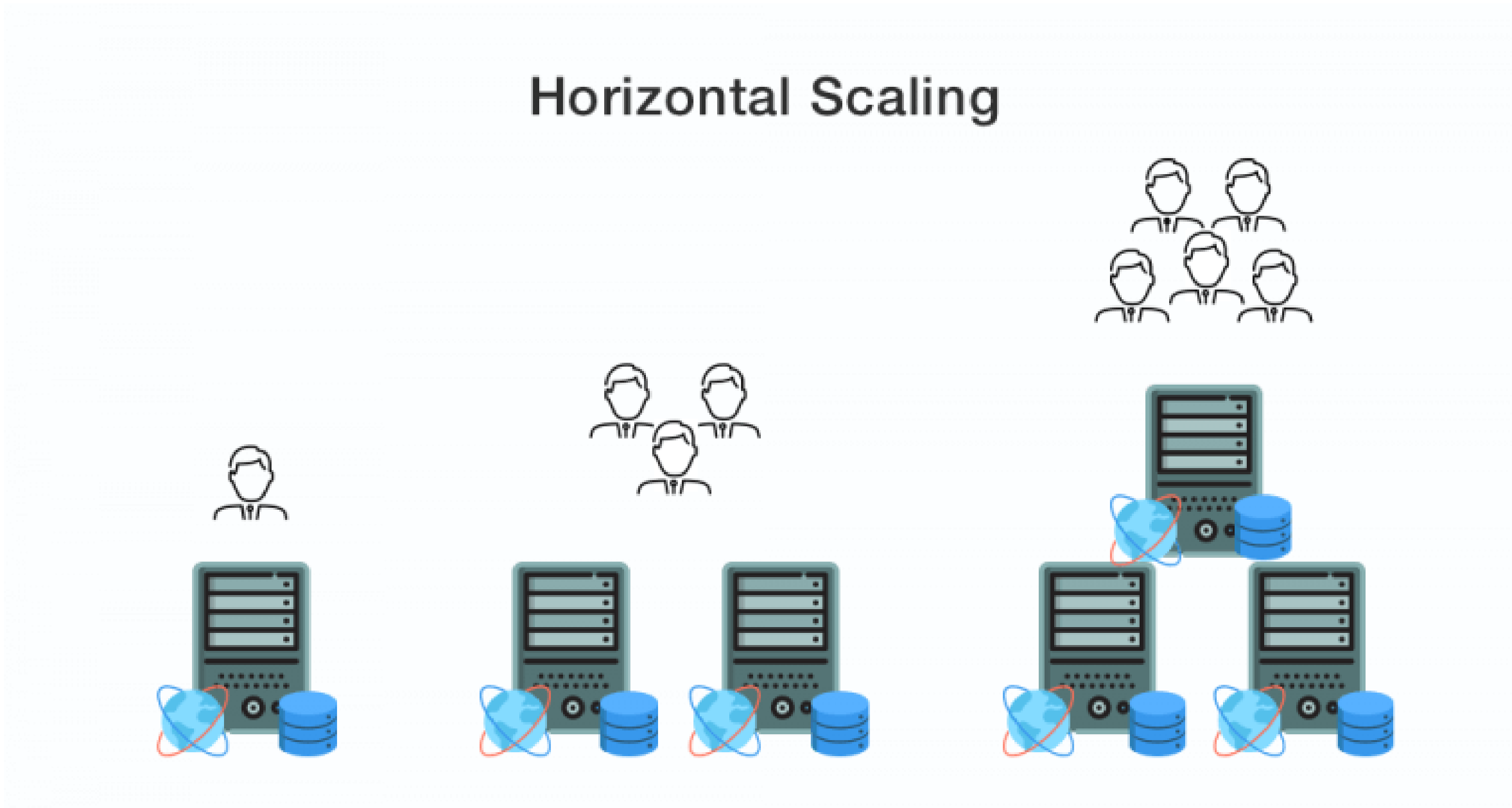
Vertical Scaling – Scale up

- increase the capacity of individual nodes, eg:
  - adding memory,
  - increasing the number of CPU cores,
  - Faster cores
  - bigger/more disks
- Advantages:
  - Simple management/deployment
  - Usually no software changes
- Disadvantages
  - Can software fully utilize hardware capacity?
  - Still limited by capacity of node
  - $$s

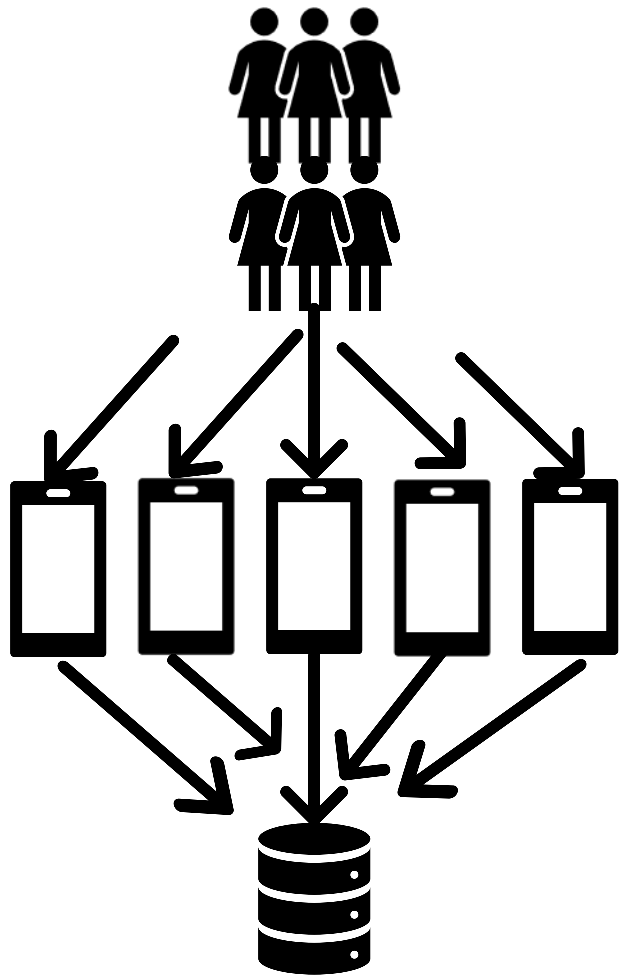
Scaling out – Horizontal Scaling

- increases overall application capacity by adding new nodes
- Node typically homogenous, eg:
  - same memory, CPU
- Advantages
  - Collective system capacity can be increased by adding more nodes
- Disadvantages
  - Requires system architecture to effectively utilize collective node capacity

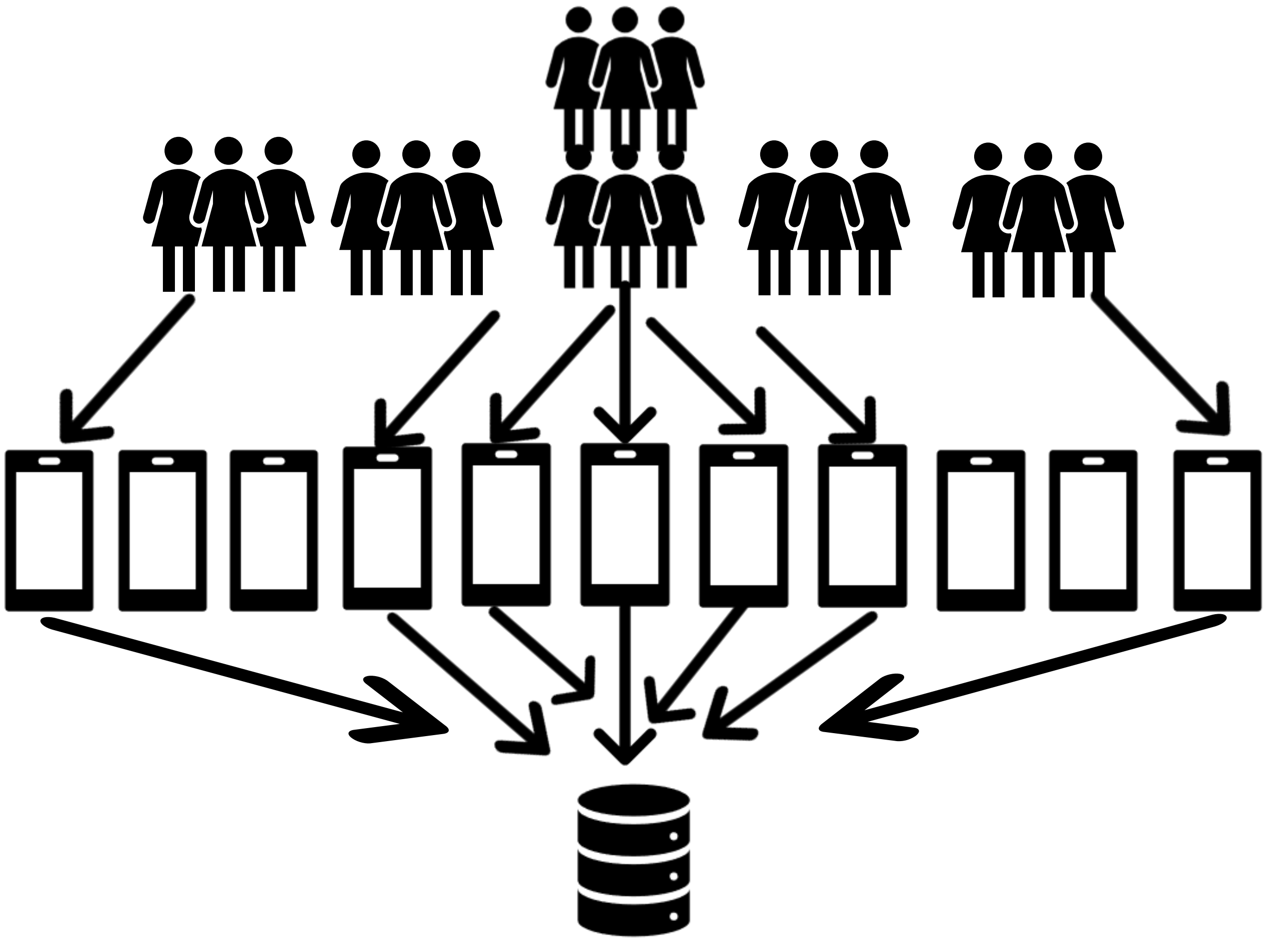
Scale Out



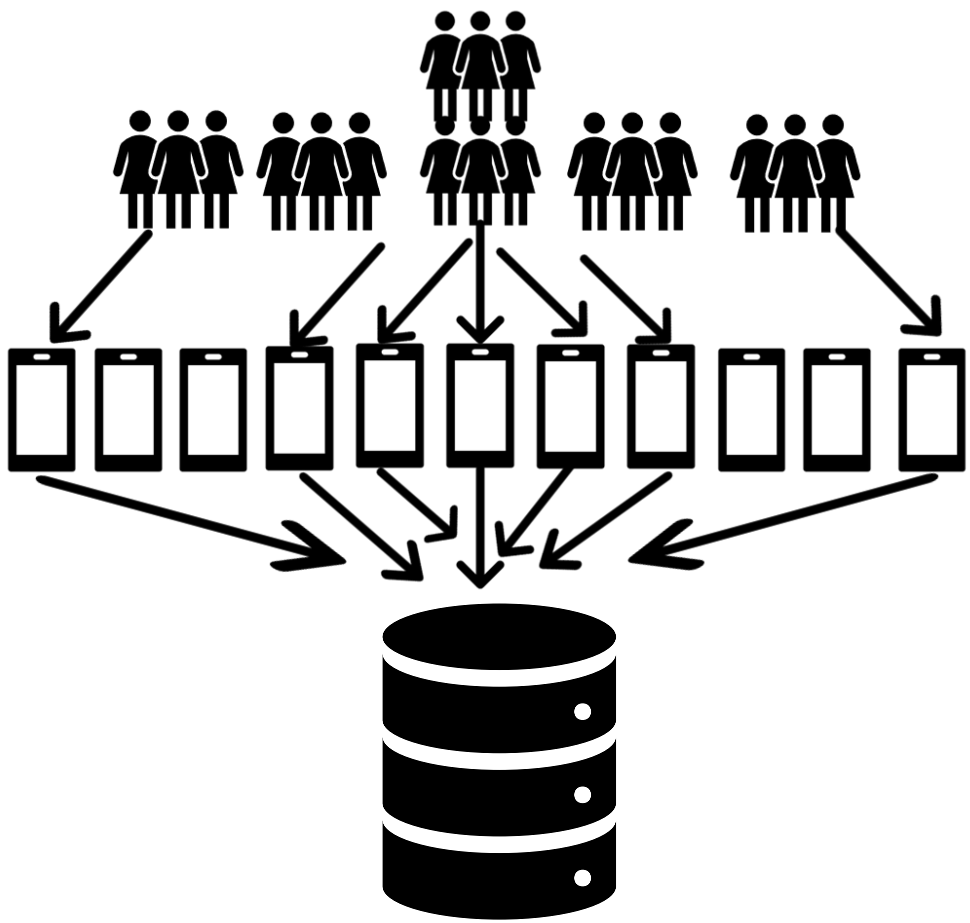
Horizontal Scaling



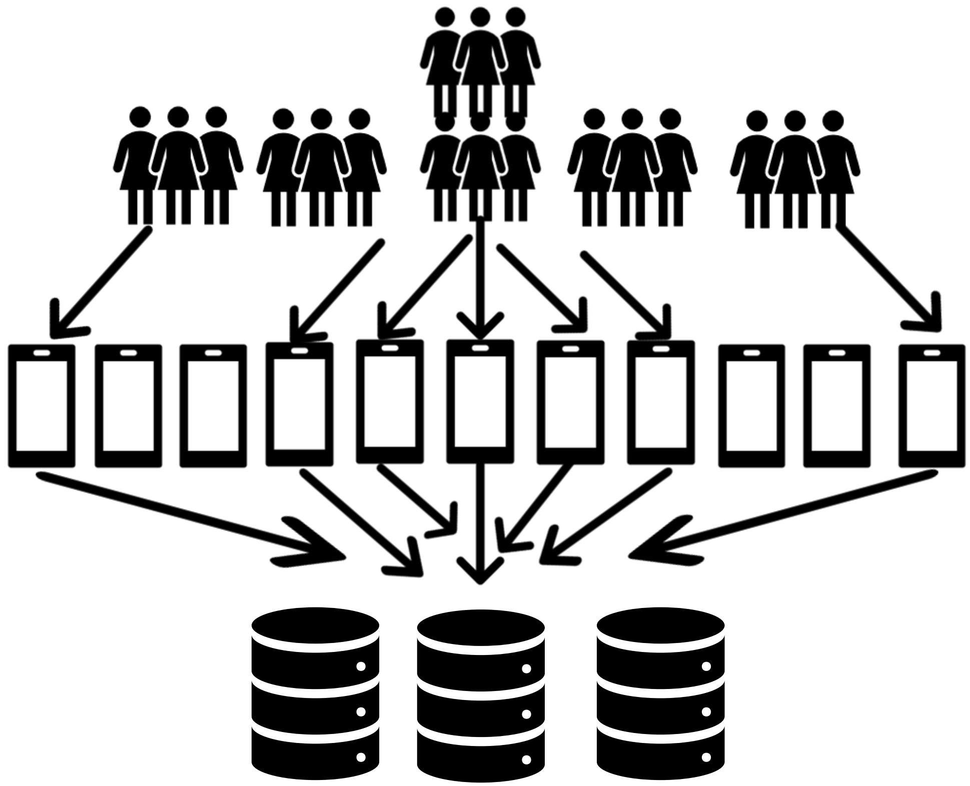
Horizontal Scaling



Horizontal Scaling



Horizontal Scaling



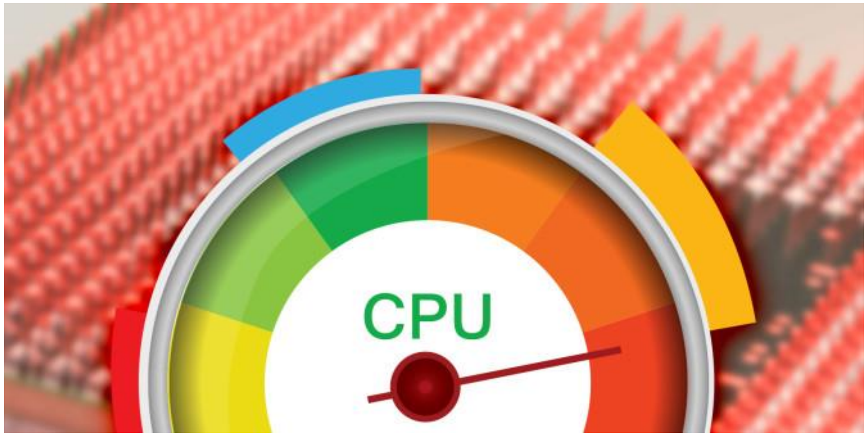
Resource Contention

- As we scale some components, we create contention in other components, eg:
  - Process more HTTP requests creates database contention
- Contention creates *bottlenecks* in our systems
- Bottlenecks have limited capacity and limit scalability
- Two options:
  - Decrease resource demand
  - Increase capacity

Decrease Resource Demand

- Introduce optimizations, eg:
  - More efficient algorithms
  - More efficient database queries
  - Less database queries
  - Compress data
  - Use a faster programming language
- Optimizations help eliminate bottlenecks by reducing resource demand
- Create ‘headroom’ so as load increases, we hit bottlenecks more slowly

Let’s examine some strategies



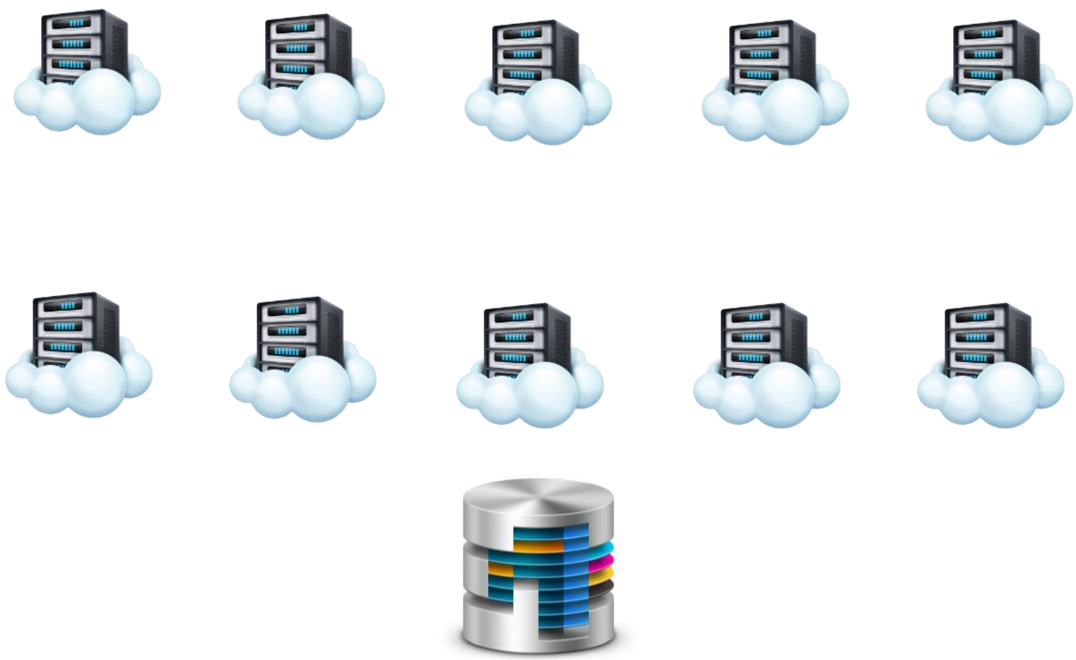
Scaling – Capacity Increases

Horizontal scaling adds resources to handle increase request loads

How do we distribute requests evenly across processing resources?

How do we handle session state?

Horizontal scaling - Architecture

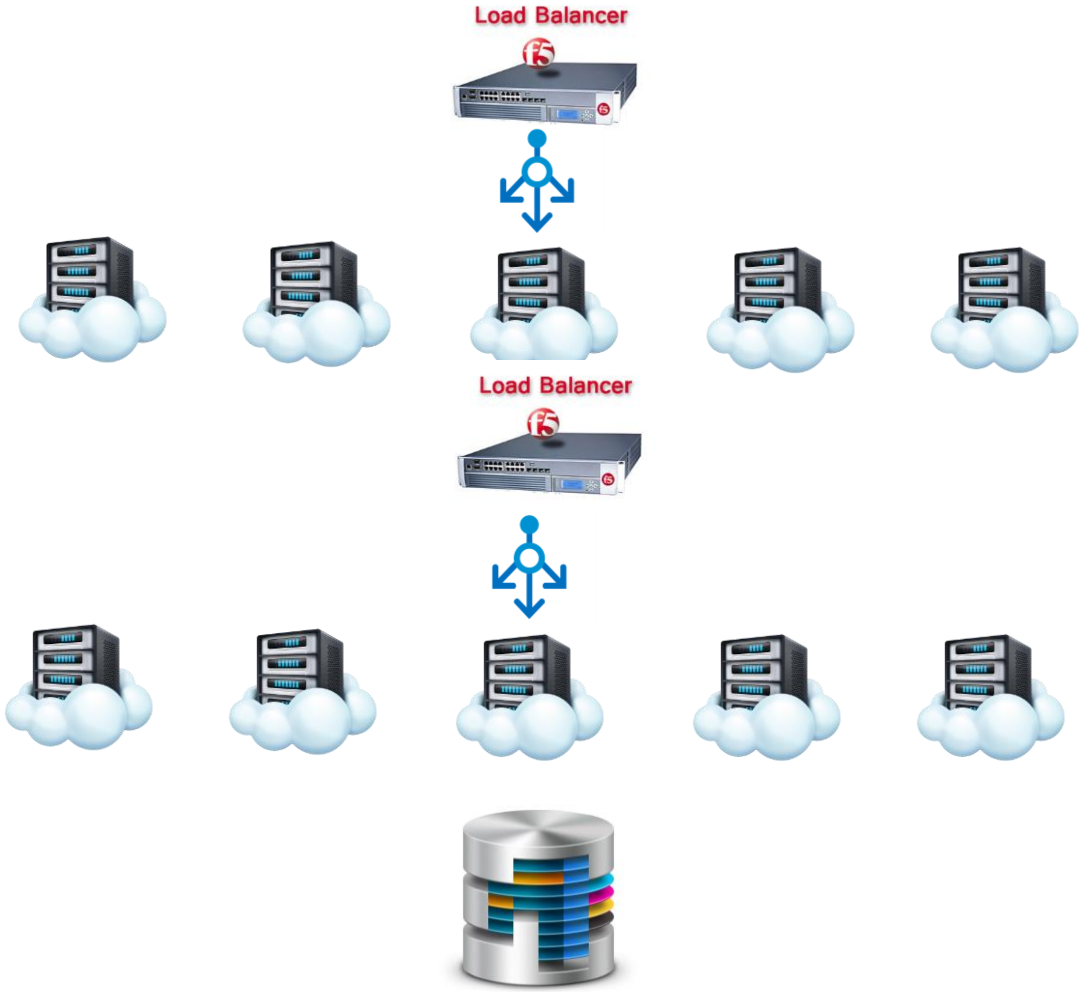


Web Server Tier

Business Logic Tier

Data Management Tier

Load Balancing



Web Server Tier

Business Logic Tier

Data Management Tier

Load Balancer

- Distributes network or application traffic across servers
- Used to increase capacity and reliability of applications
- Load balancers are generally grouped into two categories:
  - Layer 4 act upon data found in network and transport layer protocols (IP, TCP, FTP, UDP)
  - Layer 7 distribute requests based upon data found in application layer protocols such as HTTP

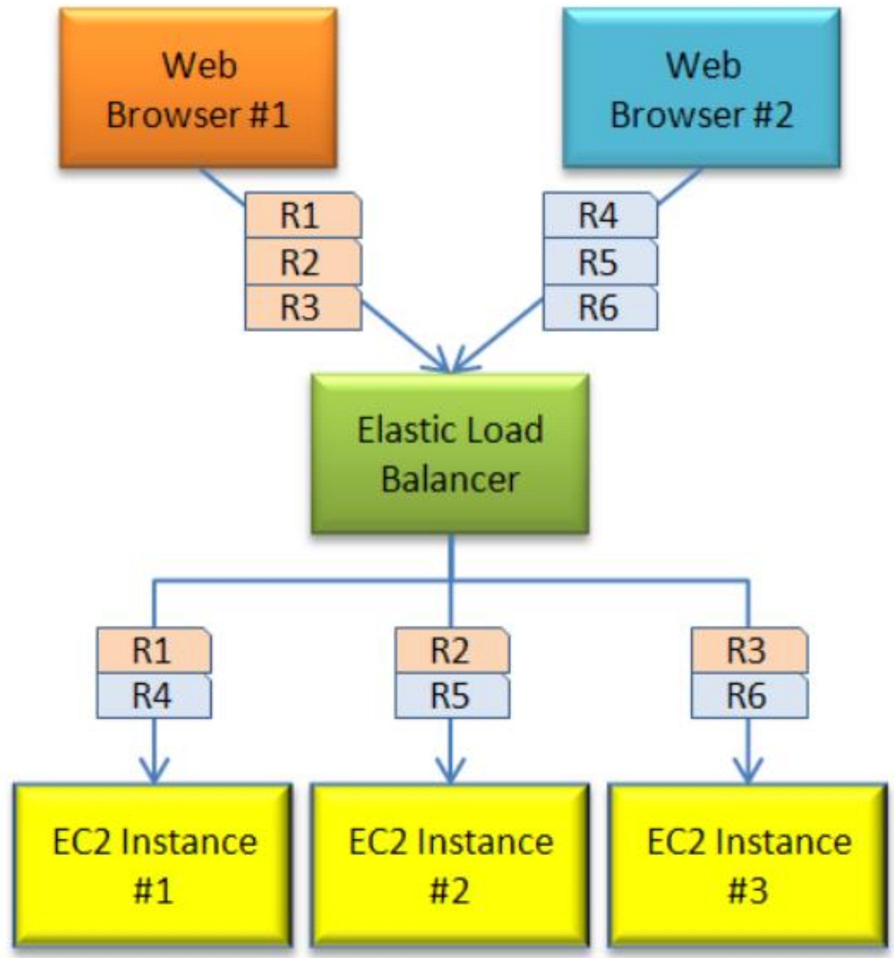
Loan Balancer Features

- Typically support various policies, eg:
  - Round robin
  - Weighted round robin
  - Least connections
  - Least response time
- Perform health checks
  - Only send to health servers
- Layer 7 can distribute based on HTTP request contents, eg:
  - Headers, cookies, parameter values,

Example: AWS Elastic Load Balancing

- Application and Network load balancers
  - https://aws.amazon.com/elasticloadbalancing/features/
- Application LB can do routing based on:
  - HTTP method
  - HTTP header field
  - HTTP query string
- Which to use?
- It’s a trade-off ;)

State Management

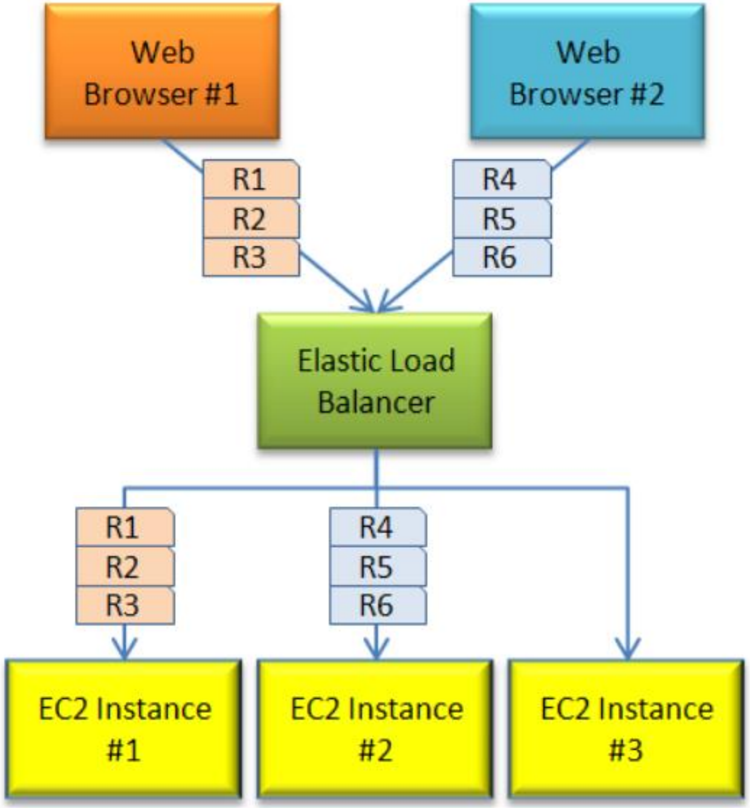


https://aws.amazon.com/blogs/aws/new-elastic-load-balancing-feature-sticky-sessions/

State Management

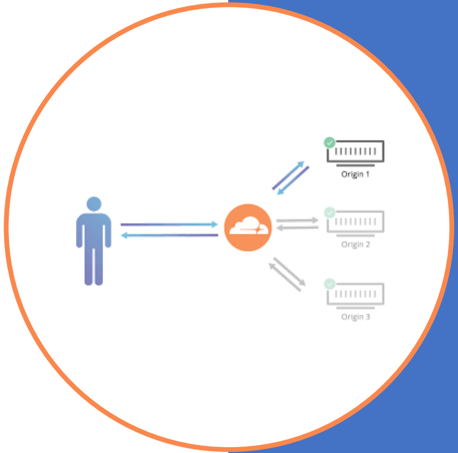
- Load balancers attempt to distribute load evenly across servers
  - Application has no (little) control over which server instance sees a request
- What if a user:
  - browse multiple pages on a site:
  - Adds some products to their shopping cart or favorites list
- Where do we store the state associated with their session?

One Solution – Sticky Sessions



Sticky Sessions

- Aka session affinity
- Server instances can cache user data locally for better performance.
- A series of requests from the user will be routed to the same EC2 instance
- If instance has terminated or failed a health check, the load balancer will route the request to another instance
  - Session data lost



Session Management

- Application generates a session ID that is returned to client
- Can be part of URL.
  - http://www.example.com/products/stuff.html?sessionID=0123456789ABCDEFGH
- Usually provided via cookies, supported by browsers
- Cookies placed in HTTP request so they can be read by the application even if a load balancer intervenes.

Cookie Example - Jersey

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response getAllEployees()
{
    Employees list = new Employees();
    list.setEmployeeList(new ArrayList<Employee>());

    list.getEmployeeList().add(new Employee(1, "Lokesh Gupta"));
    list.getEmployeeList().add(new Employee(2, "Alex Kolenchiskey"));
    list.getEmployeeList().add(new Employee(3, "David Kameron"));

    return Response.ok().entity(list).cookie(new
NewCookie("cookieResponse", “784748274283742")).build();
}
```

Cookie Example - Jersey

```
public static void main(String[] args)
{
    Client client = ClientBuilder.newClient( new ClientConfig().register( LoggingFilter.class ) );
    WebTarget webTarget = client.target("http://localhost:8080/JerseyDemos/rest").path("employees");

    Invocation.Builder invocationBuilder =  webTarget.request(MediaType.APPLICATION_JSON);
    Response response = invocationBuilder.get();

    Employees employees = response.readEntity(Employees.class);
    List<Employee> listOfEmployees = employees.getEmployeeList();

    System.out.println(response.getCookies());
    System.out.println(response.getStatus());
    System.out.println(Arrays.toString( listOfEmployees.toArray(new Employee[listOfEmployees.size()])
));
}
```

Session Management Issues

- How long do we keep the state?
  - User browses and fills up shopping cart in 5 mins
  - Doesn’t log in again for 97 days!!
- Server session cookies are one solution
  - JSESSIONID in Java
  - Default 30 mins
  - Set in web.xml

```
<session-config>
    <session-timeout>10</session-timeout>
</session-config>
```

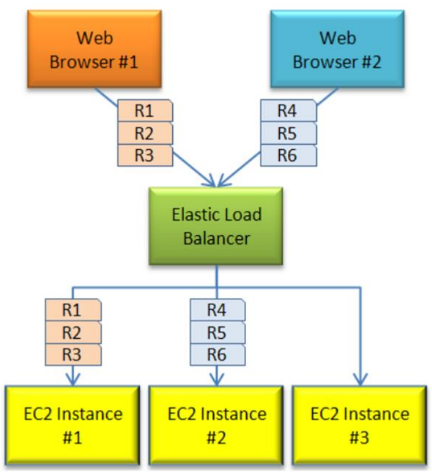
State Management Issues

- Set explicit LB timeout
- With AWS ELB

elb-create-lb-cookie-stickiness-policy myLoadBalancer --policy-name fifteenMinutesPolicy --expiration-period 900

elb-set-lb-policies-of-listener myLoadBalancer --lb-port 80 --policy-names fifteenMinutesPolicy

Timeout Duration



- Tricky
- Too short?
  - session information evaporates
- Too long?
  - Uneven request distribution possible
  - Unnecessary use of resources

Stateless Servers

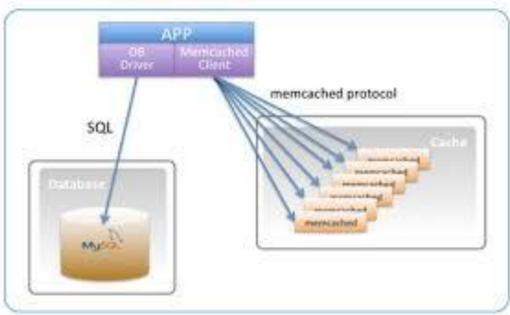
- Servers hold no session (conversational) state
- Built in fault tolerance
  - No state lost if server fails
- It's stored somewhere global that all load balanced servers can see
- Two basic options:
  - A database
  - A cache

Caching

Distributed Caches

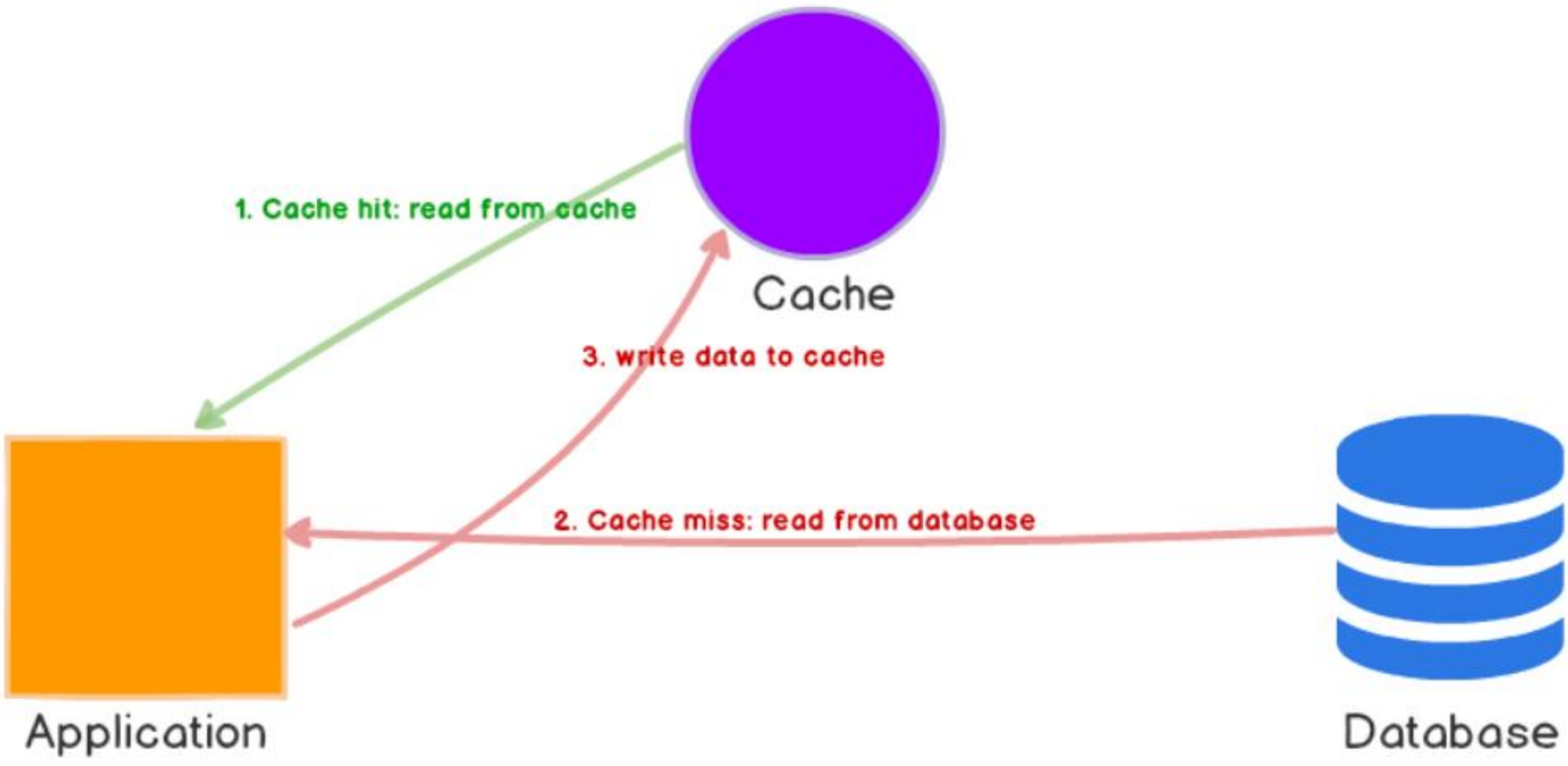
- A distributed cache spans multiple servers
- Store web session data
  - State management
- Also used to reduce database read load
  - Reducing database demand
- Feasible because memory has become cheap and network cards fast,
- 1 Gbit now standard everywhere
  10 Gbit emerging
- Many products available

Example: Memcached



- Distributed cache
- Key value store
- Distributed hash table, uses consistent hashing
- Keys 250 bytes max, values 1MB max
- Clients hash key (eg SessionID) to locate server where object may be cached
- If cache is full, keys evicted based on an LRU policy

Cache Aside


1. Cache hit: read from cache
Cache
3. write data to cache
2. Cache miss: read from database
Application
Database

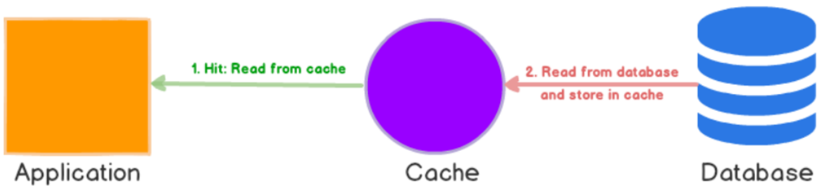
Read Example – Pseudo Code

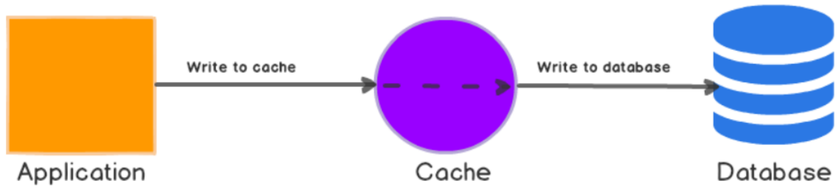
```
function get_foo(int userid)
    /* first try the cache */
    data = memcached_fetch("userrow:" + userid)
    if not data
        /* not found : request database */
        data = db_select("SELECT * FROM users WHERE
userid = ?", userid)
        /* then store in cache until next get */
        memcached_add("userrow:" + userid, data)
    end

    return data
```

Update Example

```
function update_foo(int userid, string dbUpdateString)
  /* first update database */
  result = db_execute(dbUpdateString)
  if result
    /* database update successful : fetch data to be stored in cache */
    data = db_select("SELECT * FROM users WHERE userid = ?", userid)
    /* the previous line could also look like data =
createDataFromDBString(dbUpdateString) */
    /* then store in cache until next get */
    memcached_set("userrow:" + userid, data)
```

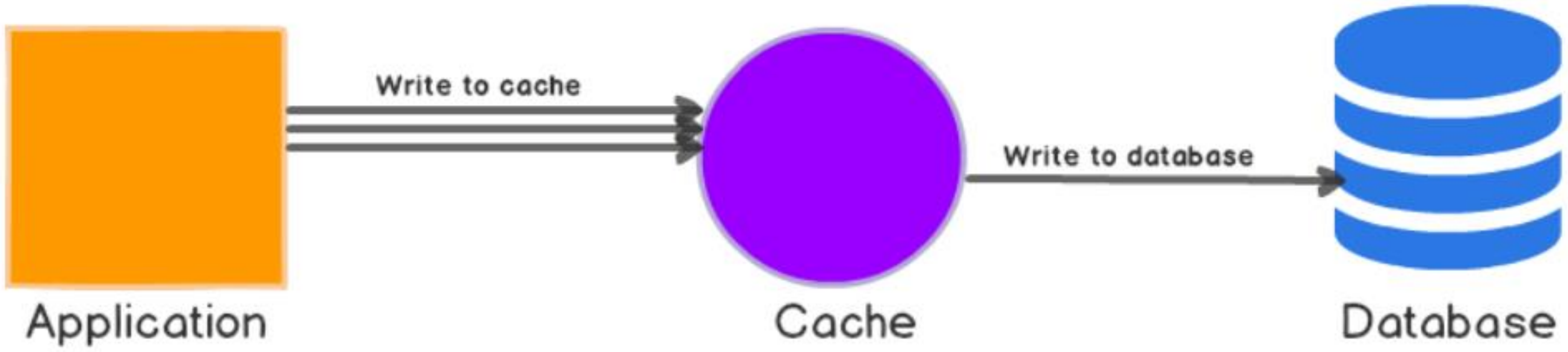




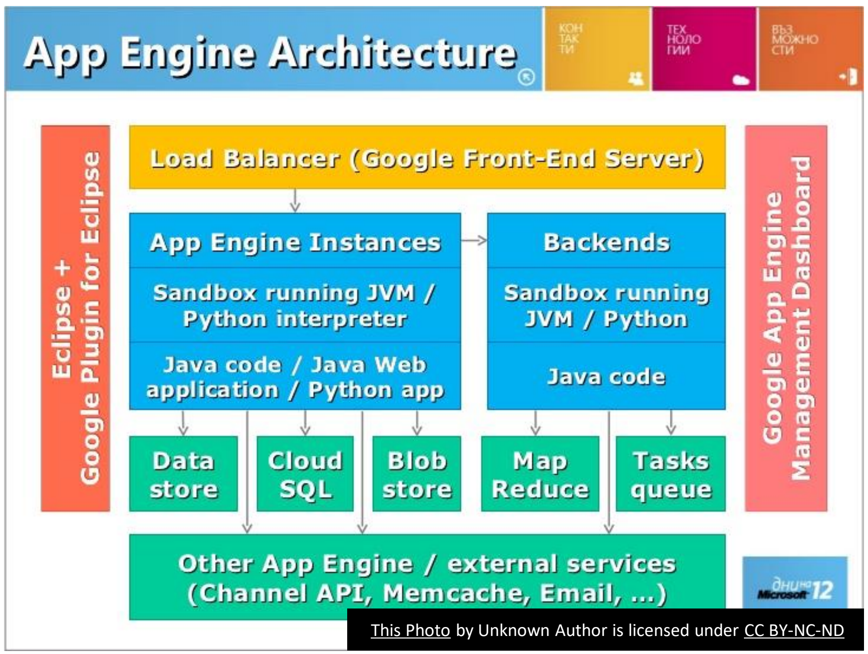
Read Through/Write Through Database Caches

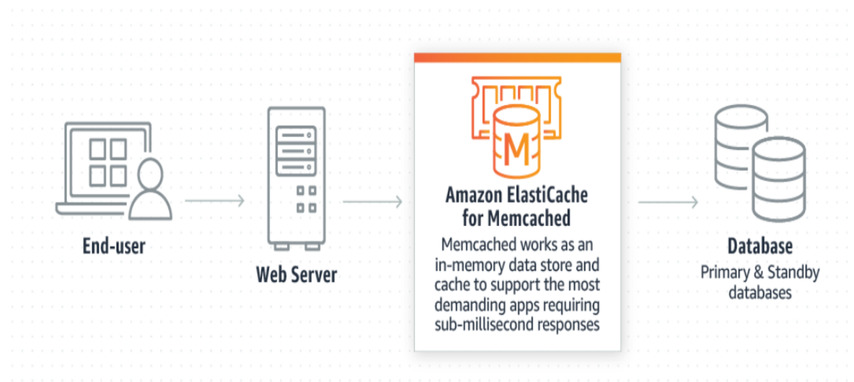
Write Back/Write Behind Cache

Commonly enabled in databases by default

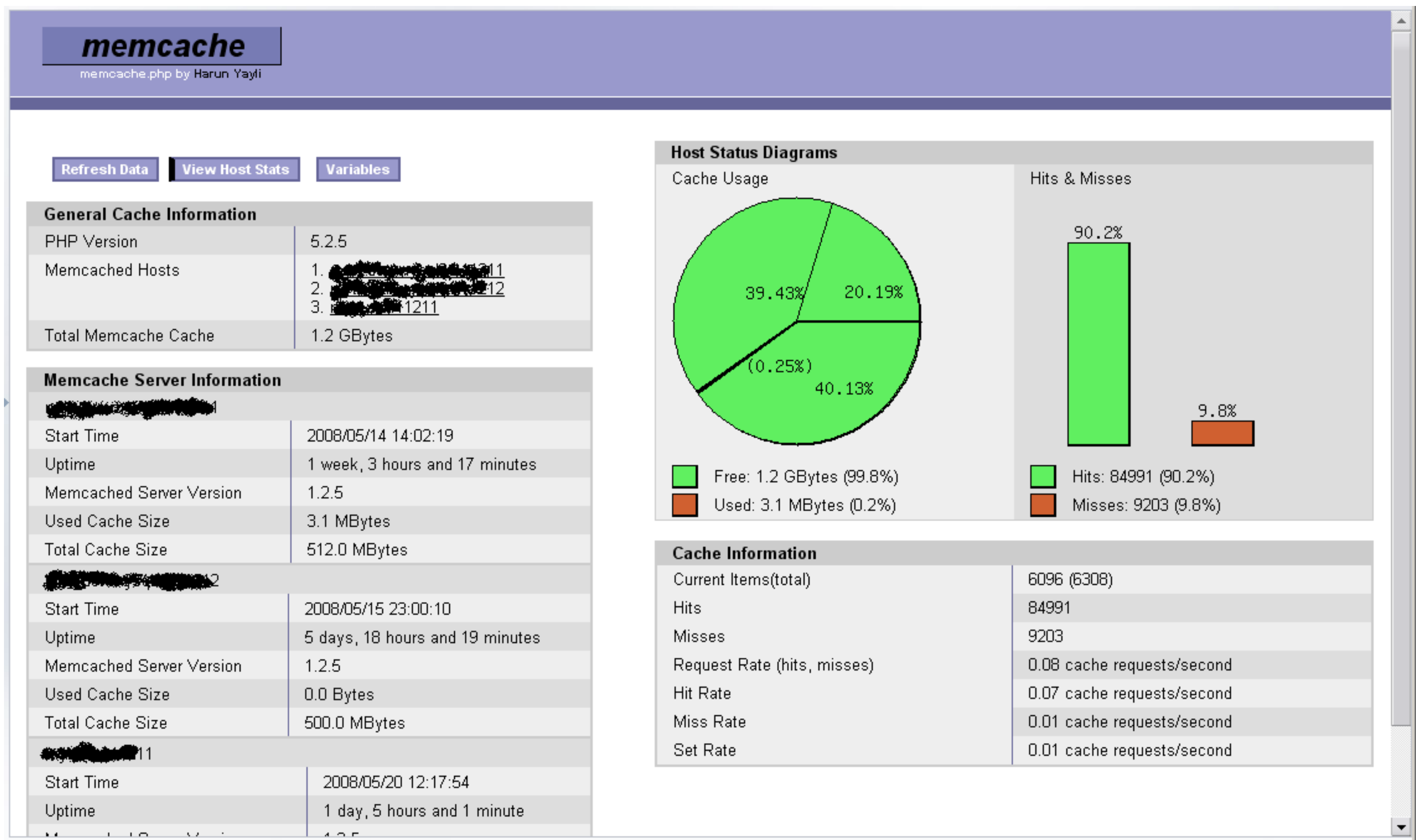


Cloud Services

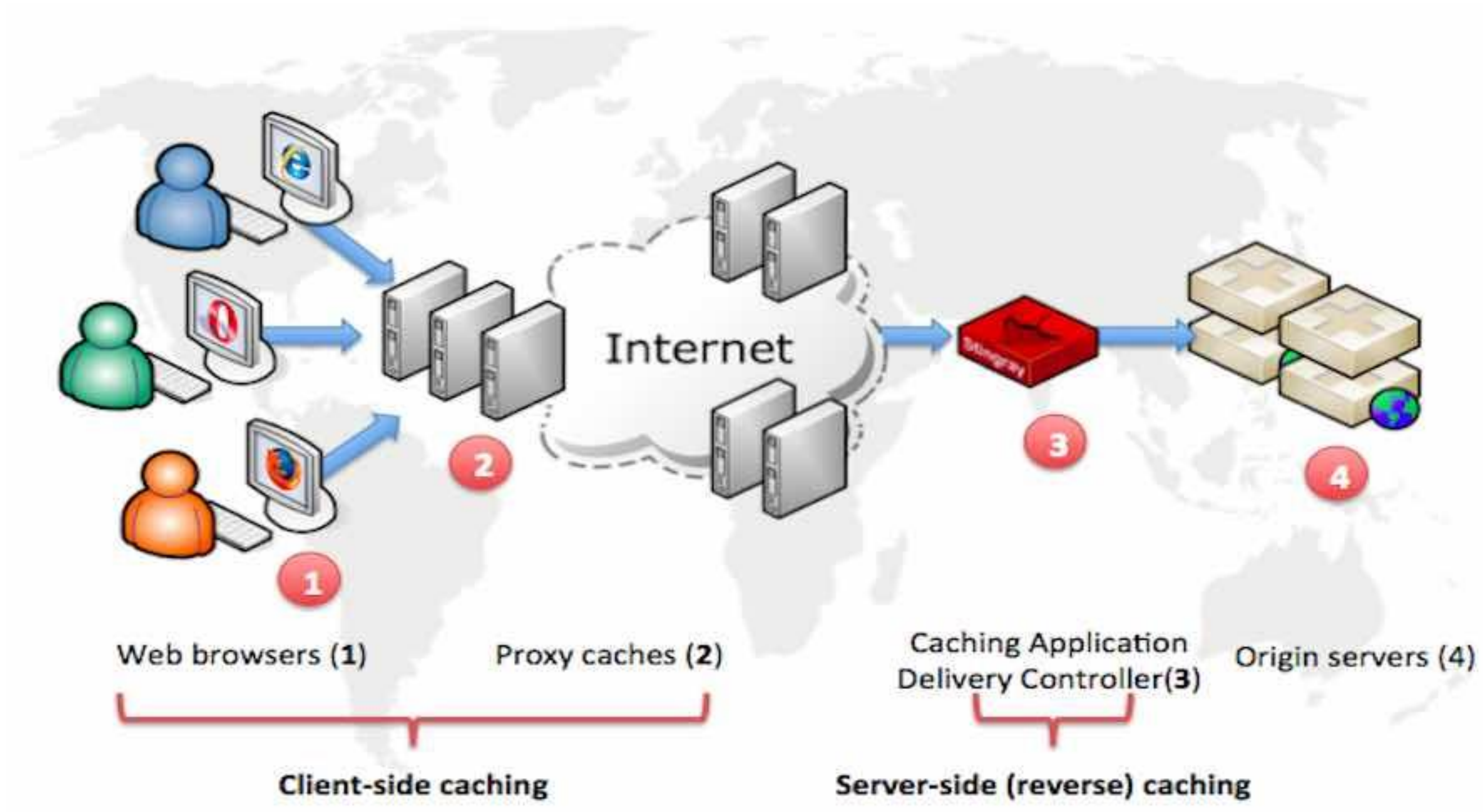




Caching Effectiveness



Web Caching



Internet
Stingray
1
2
3
4
Web browsers (1)
Proxy caches (2)
Caching Application
Delivery Controller(3)
Origin servers (4)
Client-side caching
Server-side (reverse) caching

Web Caching

- store copies of frequently accessed data in caches along the request-response path.
- requests go through a cache or a series of caches toward the service hosting the resource.
- Any caches along the request path with a fresh copy of the requested representation can respond to the request
- If no cached content, the request is served by the API service (origin server).

HTTP Caching Directives

- HTTP has various directives to control caching
  - Both client and server side
- Servers can set these in response headers to control how content is cached and accessed in the WWW
- By default, GET responses may be cached
  - With directives we can add much more control over how this happens

HTTP Caching Directives: Examples

Expires

- Absolute time

Last-Modified

- Date resource last changed

Etag

- opaque string that a server associates with a resource to uniquely identify the state of the resource. When the resource changes, the entity tag changes.

Cache-Control

- max-age=<delta-seconds>
- public/private
- must-revalidate/proxy-revalidate
- no-cache/no-store
- < …more…>

Example

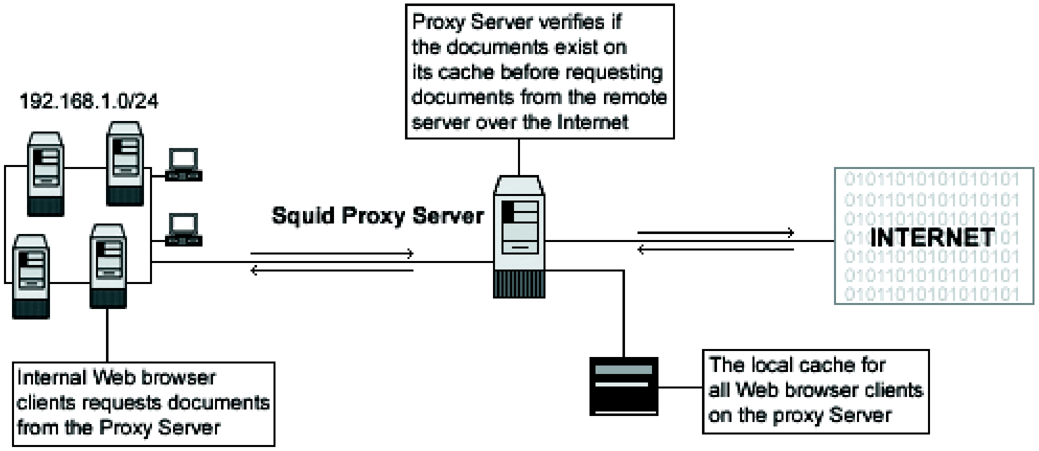
Request:
GET /upic.com/liftlines/Blackstone

Response:
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 26 Mar 2019 09:33:49 GMT
Expires: Fri, 26 Mar 2019 09:38:49 GMT

<!-- Content omitted -->

Then caching magic happens

- ***Caches such as Squid/Varnish handle this behavior for us for free.***
- ***Consumer applications don’t need to take any notice of ETag and Last-Modified values: validations are dealt with by the underlying caching infrastructure.***



Example: Daily Weather Report

Each morning a weather reporter for each ski resort posts a summary of the conditions/weather, with

- Open lifts
- Images from web cam
- Other resort specific stuff

These reports are changed maybe once or twice a day

- Sometime never
- Irregular changes (reporter may be skiing!!)
- More often is an active weather day

Example

Request:
GET /upic.com/weather/Blackstone

Response:
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/json
Date: Fri, 26 Mar 2019 09:33:49 GMT
Cache-Control: public, max-age=3600
ETag: “09:33:49 **"**
<!-- Content omitted -->

….. 2 hours later, no changes to report…..

Request:
GET /upic.com/weather/Blackstone
If-None-Match: " 09:33:49“

Response:
HTTP/1.1 304 Not Modified

How do we implement the server?

- Needs to determine if weather report changed from etag …
  - Lightweight
  - Fast
- Look up in a database?
- Look up in cache?

One possibility

Generate new report

- Build report and store in a database
- Create new cache entry {#resortname-weather, etag value}
- Serve report to clients and pass etag

When conditional requests arrive

- Lookup etag value in cache at {#resortname-weather}
- Return 304 if etag the same
- Serve new report and new etag if not

Update report

- Build updated report and store in a database
- Update cache entry {#resortname-weather, new etag value}

Summary

- Scaling means adding capacity and/or reducing demand
- Horizontal scaling with stateless services adds capacity
- Caching reduces demand on data tier
- HTTP has many useful mechanisms to exploit inbuilt cache infrastructure