Day 10

This is CS50 for MBAs. Harvard Busines School. Spring 2015.

Cheng Gong

Table of Contents

Requests

- In the command prompt (or Terminal), we can type a command called `ab` that sends HTTP requests and times how long it takes to request a response.
- One run might look like this:

```
% ab -n 1 http://www.harvard.edu/
This is ApacheBench, Version 2.3 <$Revision: 1604373 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking www.harvard.edu (be patient).....done


Server Software:        Apache
Server Hostname:        www.harvard.edu
Server Port:            80

Document Path:          /
Document Length:        60245 bytes

Concurrency Level:      1
Time taken for tests:   0.076 seconds
Complete requests:      1
Failed requests:        0
Total transferred:      60924 bytes
HTML transferred:       60245 bytes
```

```
Requests per second:    13.22 [#/sec] (mean)
Time per request:       75.657 [ms] (mean)
Time per request:       75.657 [ms] (mean, across all concurrent requests)
Transfer rate:          786.39 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       17   17   0.0      1       1
Processing:    58   58   0.0     58      58
Waiting:       20   20   0.0     20      20
Total:         76   76   0.0     76      76
```

# Basically, this command is pretending to be a browser and asking for Harvard's home page, and one important number is `Requests per second`, which we got a value of `13.22` for. But since this is based on only one test, it's not very reliable.

- We can use `-n 10` for 10 requests this time, as opposed to 1:

```
% ab -n 10 http://www.harvard.edu/
...
Concurrency Level:      1
Time taken for tests:   1.047 seconds
Complete requests:      10
Failed requests:        0
Total transferred:      609231 bytes
HTML transferred:       602450 bytes
Requests per second:    9.55 [#/sec] (mean)
Time per request:       104.689 [ms] (mean)
Time per request:       104.689 [ms] (mean, across all concurrent
 requests)
Transfer rate:          568.30 [Kbytes/sec] received
...
```

- We see that it's actually a little slower this time, so let's try with 100 requests:

```
% ab -n 100 http://www.harvard.edu/
...
Concurrency Level:      1
Time taken for tests:   10.776 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      6092285 bytes
HTML transferred:       6024500 bytes
```

```
Requests per second:    9.28 [#/sec] (mean)
Time per request:       107.760 [ms] (mean)
Time per request:       107.760 [ms] (mean, across all concurrent
 requests)
Transfer rate:          552.11 [Kbytes/sec] received
...
```

- Right now this program is actually single-threaded, which means that it can only do one thing at a time, in this case making one request at a time and waiting for it before it can make another one.
- Looking at our results, we see we're doing a little worse. So we can speed things up by running the program with a `-c` option, or the multi-threaded version that uses multiple cores of the CPU to run what we can think of as multiple copies of the program.

```
% ab -n 100 -c 10 http://www.harvard.edu/
...
Concurrency Level:      10
Time taken for tests:   2.647 seconds
Complete requests:      100
Failed requests:        54
    (Connect: 0, Receive: 0, Length: 54, Exceptions: 0)
Total transferred:      6094035 bytes
HTML transferred:       6026390 bytes
Requests per second:    37.78 [#/sec] (mean)
Time per request:       264.709 [ms] (mean)
Time per request:       26.471 [ms] (mean, across all concurrent requests)
Transfer rate:          2248.20 [Kbytes/sec] received
...
```

  # Now the program can send 10 requests at a time, in parallel. And we see a much higher count of requests per second.

- Let's try running 1000 requests with sending 100 at a time:

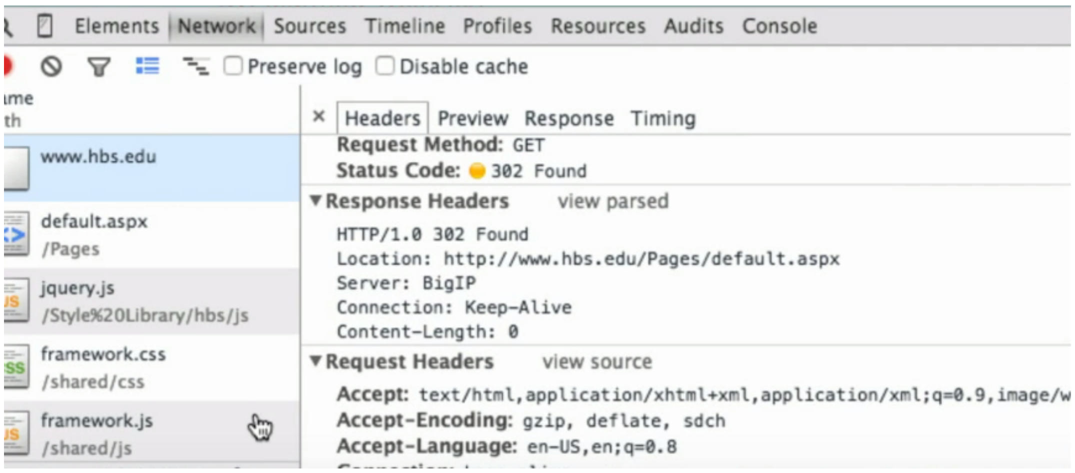
```
% ab -n 1000 -c 100 http://www.harvard.edu/
...
Requests per second:    190.28 [#/sec] (mean)
...
```

- So this information is useful for helping us figure out how many users a given server can handle per second.

- Let’s try it with HBS' server:

```
% ab -n 1000 -c 100 http://www.hbs.edu/
...
Non-2xx responses:      1000
Requests per second:    2700.73 [#/sec] (mean)
...
```

- That was really fast, possibly partially because we’re on campus and a little closer to the servers. But the line above, `Non-2xx responses`, indicates that all 1000 of our requests did not result in a proper response. Remember that HTTP status code 200 means "OK", so a non-2xx response is probably indicating some error, like 404.
- If we open our browser and go to `http://www.hbs.edu`, we see that we get redirected to `http://www.hbs.edu/Pages/default.aspx`, so the response from the server was probably redirecting us. In fact, we can look with Chrome’s inspector:



```
www.hbs.edu
default.aspx /Pages
jquery.js /Style%20Library/hbs/js
framework.css /shared/css
framework.js /shared/js
```

```
Request Method: GET
Status Code: 302 Found
▼Response Headers
HTTP/1.0 302 Found
Location: http://www.hbs.edu/Pages/default.aspx
Server: BigIP
Connection: Keep-Alive
Content-Length: 0
▼Request Headers
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/w
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

# The very first response was indeed `302`, with the correct location sent back.

- So the website seemed really fast because all it was returning was the location to redirect to. So let’s run the command with the actual URL:

```
% ab -n 1000 -c 100 http://www.hbs.edu/Pages/default.aspx
...
Requests per second:    70.43 [#/sec] (mean)
...
```

- Now we’re at a much slower 70 requests per second. Looking back at the actual homepage, we see lots of graphics and images and videos, so the website might be pulling information from a database or rendering the page every time, and that's where we might start optimizing for more requests per second.

- The redirection, too, is problematic for people on slower internet or mobile connections, since each request might take some time, and one is basically wasted on redirecting to another page.

Questions

- I'm still struggling to understand the concept of a callback function. Could we go over that again?
  - A synchronous call would be like David calling Daven on the phone, where David asks Daven a question like "What time is it?" Daven takes a few seconds to look this up, and while he's looking it up, David can't do anything except stay on the phone. An asynchronous call would be like David calling Daven to ask the question, telling him to reply when he knows the answer, and hanging up right away. In this case, David could be doing something else in the meantime, and Daven would be the callback function when he literally calls David back. In programming, a callback function would also interrupt a running program to do stuff. JavaScript is generally single-threaded on the client side, so it can only do one thing at a time, which is why asynchronous functions are important.
- So when we talked about Ajax and you showed us the lag Google maps exhibits when dragging around quickly, it seems what actually happens is that for a split second we are shown a very blurry image of a map, which then comes into focus a second later. It seems that the lag is not due to communication time with the server (as I presume we would see a blank space for a second instead of seeing a blurry map that eventually comes into focus) but rather the rendering time on the client side. Is this a correct way to think about it? In general, when it comes to client-side image rendering, does that lag time usually outweigh the lag of the actual communication time with the server? Is that a bottleneck of the browser's power or of the client-computer's RAM?
  - It's true that some of the delay is due to the computer rendering the image, but modern computers are fast enough that this happens almost instantaneously. The "blurry image" is due to the server sending the image in pieces with increasing levels of detail, which is why those details appear after some delay.
- When the appearance/format of a website changes dynamically (for example a carousel graphic, or elaborate iterations of graphics or headers, for instance here http://

gisele.underarmour.com) is this an example of ajax or asynchronization? Or is this a different complicated front-end programming language and ajax is more back-end?

  # The website (which is no longer accessible), probably had a video playing in the background, and without digging deeper hard to say if it was downloaded as the page loaded, or via Ajax a little bit at a time (like YouTube's video buffering). Ajax is definitely front-end, however, since it's code running on the client side, talking to a server and getting data from it. A carousel of images, too, might be implemented either with all the images downloading at once, or only when you click the left or right arrows, in which case Ajax would be used to download the image from the server as we need it.

- What's the difference between local storage and a cookie? When is an example you would use local storage and not a cookie?

  # A cookie is sent back to the server every time you visit a page in that domain, and local storage is only stored on the user's browser. An example would be using local storage to temporarily store a shopping cart, for example, and not sending it back to the server every time another page is visited, but only on the checkout page.

Scalability

- Suppose we want to increase the number of simultaneous users we can support. There are generally two solutions on the hardware side: vertical scaling and horizontal scaling.
- Vertical scaling is increasing resources on one server, by upgrading its CPU, disk, or RAM to have bigger capacities. The downside to this is both the price and the fact that technologies continue to be released constantly, so we need to upgrade often to have the best. And having just one server is risky in that we have just one point of failure.

  # One technology is the use of SSDs, or solid state disks that are all electronic flash memory disks that have no moving parts, so they are faster and more durable. But they are smaller and more expensive compared to traditional hard drives.

  # Another technology is RAID, where various numbers of drives can be configured to store data, typically duplicated in some way, to add performance or reliability. In RAID 0, for example, half of data is stored on one drive and half is stored on another, so reading and writing a chunk of data will theoretically be twice as fast, but either drive failing means half (in practice all) the data will be lost. For RAID 1, the same data is written to both disks, so if one fails, there is no effect on the system. And

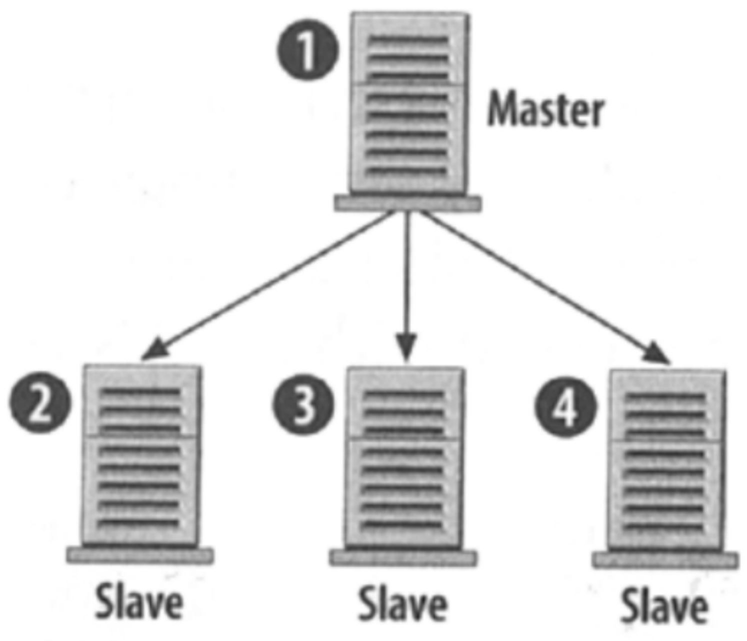
  there are combinations that give the performance and redundancy of both these methods, at the cost of requiring even more separate drives.
  - More RAM is better because it's much faster than SSDs, and it stores programs that are running, so the more RAM, the more programs can be run simultaneously without slowing down. Otherwise, programs need to be swapped from disk (slow) to memory (fast) back and forth.
- Horizontal scaling is the logical alternative, where we acquire multiple entry-level machines, for a higher total amount of resources. But the issue here is balancing users across these resources, since we have many servers sharing a common load.
  - Sometimes we'll see websites redirecting users to domains that begin with `www1.` or `www2.` or the like, and that might indicate a different server that they have, but in these cases some users might bookmark that address and always go to that server, which doesn't really spread the load out equally. Even worse, we might eventually have fewer or more servers, so the user might not be able to get on a site at all.
- Another command one might run would be `nslookup`:

```
% nslookup www.google.com
Non-authoritative answer:
Name: www.google.com
Address: 173.194.219.99
Name: www.google.com
Address: 173.194.219.106
Name: www.google.com
Address: 173.194.219.147
Name: www.google.com
Address: 173.194.219.104
Name: www.google.com
Address: 173.194.219.105
Name: www.google.com
Address: 173.194.219.103
```

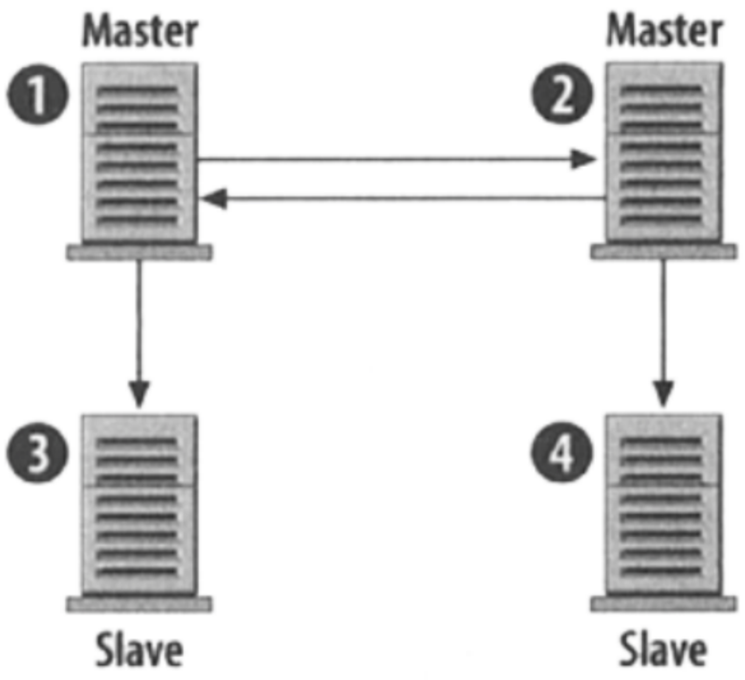
- This gets us the IP address associated with a domain, but in Google's case, there are many addresses. We see 6 servers here, but that's probably not the extent of how many servers they have. It's possible that these are the local servers, or the ones closest to the DNS server we asked, or perhaps different results will be returned at different times based on the load.
- Going back to horizontal scaling, having lots and lots of servers (hundreds or thousands) creates lots of problems, like power, heat, and maintenance. (A data center

with lots of hard drives, each with a small chance of failure, for example, will have to deal with a few failed drives every day.)

- Caching in general is the idea of storing some data temporarily, instead of rebuilding or recalculating the results over and over again, which is slower.
  - Python, usually an interpreted language, can be compiled on the server-side into bytecode, which is faster to interpret than interpreting Python but still slower than object code that a computer understands directly.
  - MySQL also allows for the enabling of a query cache, or saving the results of requests in memory, so common requests are much faster. But when the original data changes, we need to remove the incorrect results from the cache.
  - Memcache uses memory to store generated data like HTML code.
  - Redis is similar, but object-oriented, so JSON objects can be stored easily.
- Load balancing is another issue, that this graphic might represent:
- Between users and the servers, we have this box that we call a Load Balancer, which is either a software program or a piece of hardware that divides users among servers that have content.
- DNS was already a type of this, where we can provide different IPs to different users. But DNS is cached by various places such as browsers and ISPs, so reactions to sudden changes in traffic, like changes to the numbers of servers that we have, won’t be reflected immediately, so users might not have the optimal experience.
- We also have the issue of sticky sessions, or the use of cookies and local storage to keep users' data or to keep them logged in. But if cookies are tied to information stored on one server, and we get a different DNS result later that sends us to another server, that might break our saved sessions, since the new server doesn’t know about our session.
- We could have a central server that keeps shared state, but that reintroduces the weakness of having one point of failure again. We could also have all the servers that respond to users talk to each other somehow, but that becomes messy once we have many servers all needing to exchange information with each other.
- One way to solve this is to have a central database duplicated among many physical servers, in what is called master-slave replication:



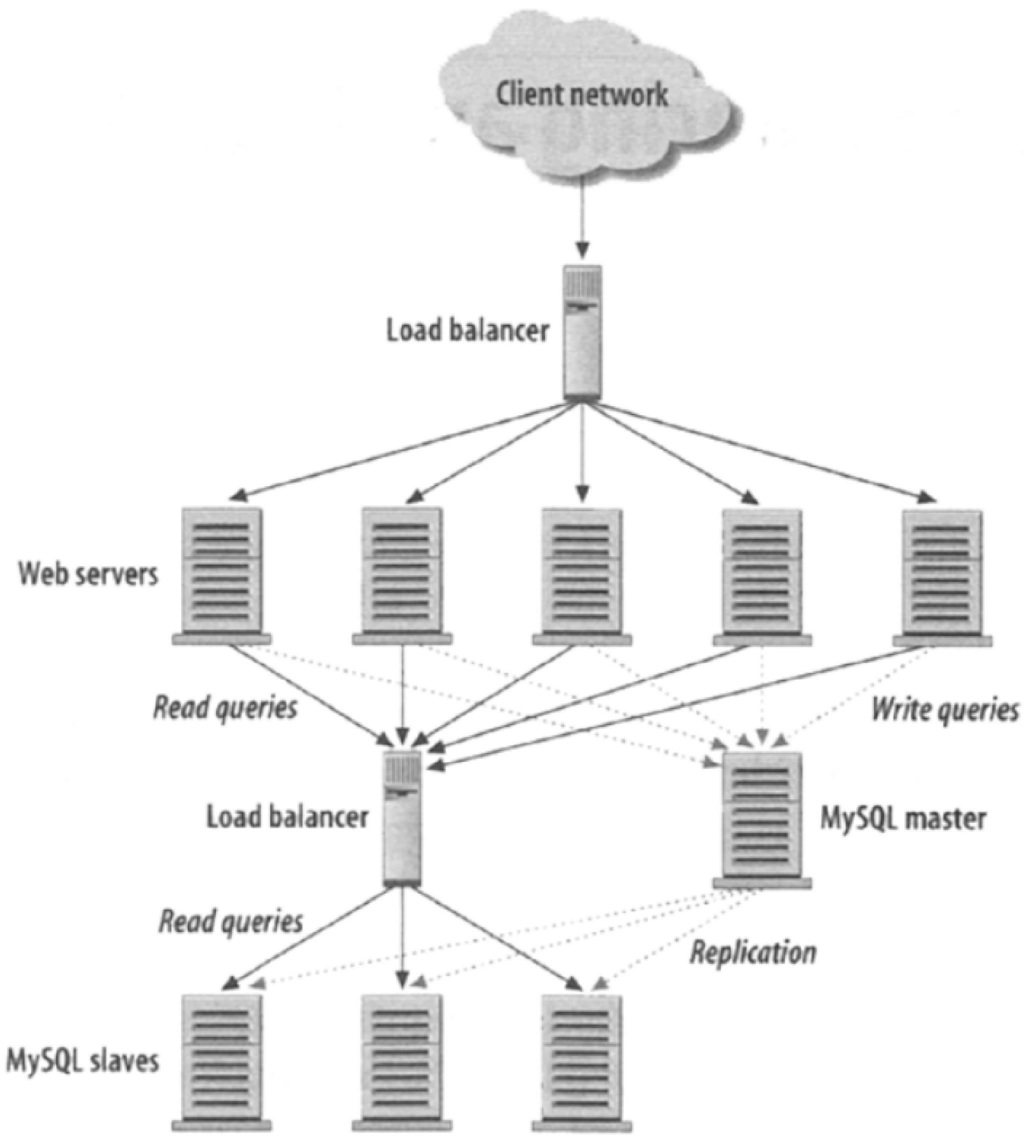
- In this case, we store all our data in a master, or primary server, and it automatically sends information to all of the slaves, or secondary servers.
- If the master server fails, then one of the slaves automatically takes over.
- Another advantage, apart from redundancy, is performance, because now all the front end servers have many of these back-end servers to talk to for reading.
- We can do even better by implementing master-master replication, where we have multiple master copies of our data, though this is challenging to program:



# Some databases, like MySQL, come with the ability to turn on this feature.

# And one master server failing could still mean that data is lost, if it didn't have the chance to pass the changes along to the other master server.

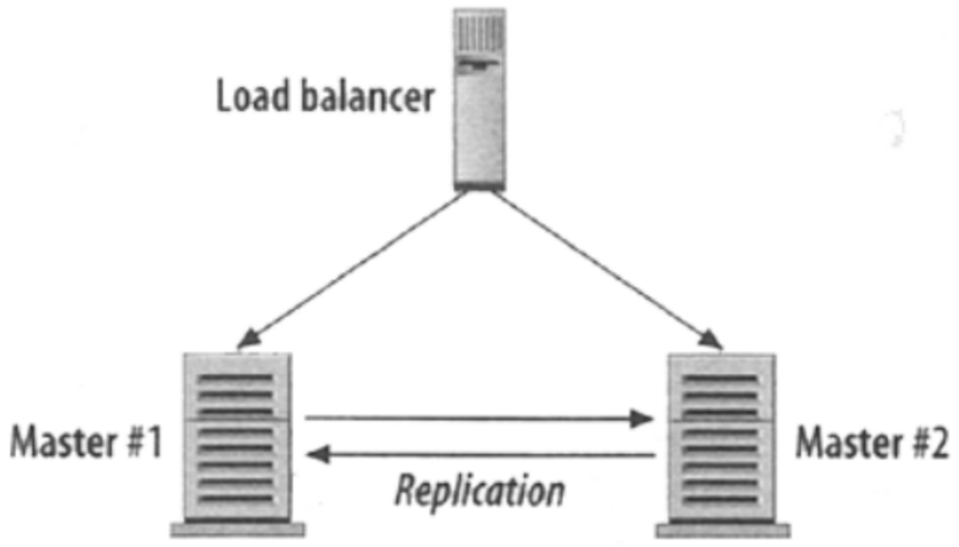
- One common complete solution is as follows:



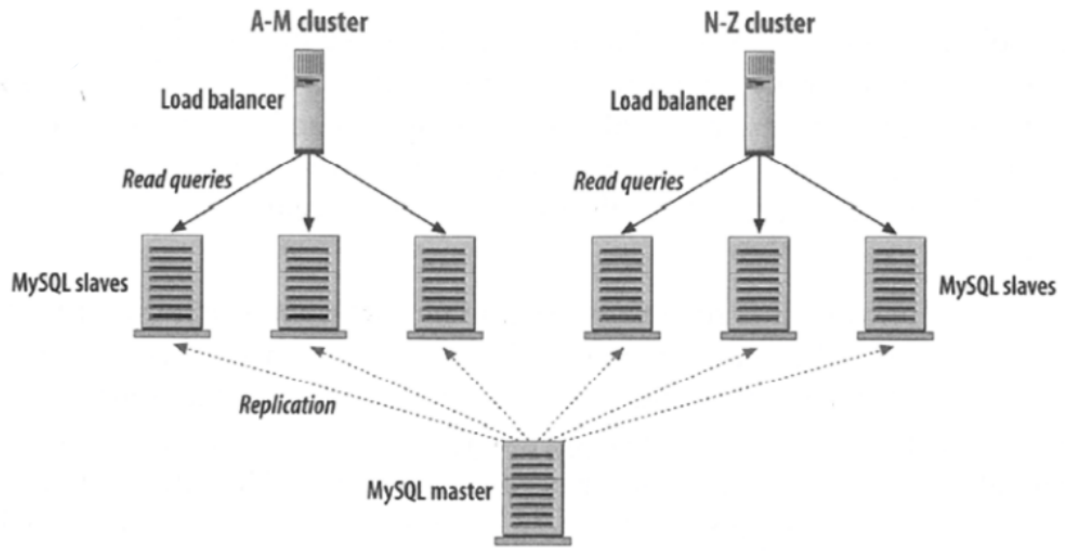
# Requests from the clients are balanced between the front-end web servers, whose read requests go to a load balancer that distributes them among many back-end slave servers, and all writes go to one or more masters that sync with the slaves.

# But we still have a single point of failure with just one device balancing the load. We can fix this by having multiple load balancers, each with a different IP, and using DNS to distribute those IPs.

# Another solution is to have two idential load balancers, one being primary and the other checking on its status, and only taking over when the first is down.



- We can also partition users' data between servers, instead of having many complete copies of data:



  # In this simple example, users with names A-M have their data stored on some servers while users with names N-Z have another set, and only one master server has all the data.

- In any case, nowadays the "cloud" solves many of these problems for us. Some examples include:

  # Amazon Web Services

  - # Google App Engine
  - # Microsoft Azure
  - # …
- Amazon, in particular, has a long list of services:
  - # CloudFront
  - # Elastic Compute Cloud (EC2)
  - # Elasticache
  - # Elastic Load Balancer (ELB)
  - # Glacier
  - # Relational Database Service (RDS)
  - # Route 53
  - # Simple Storage Service (S3)
  - # …
- Using EC2, for example, is like renting someone else's servers. But we can choose specific specs and change them with some flexibility, which is much better than buying and maintaining your own servers.
- We can see EC2 pricing[1] here, which is relatively cheap and simple at pennies per hour, as opposed to purchasing hardware, data center space, electricity, maintainance, etc.
- ELB routes requests to multiple servers, and Route 53 provides a DNS service.
- CloudFront replicates and caches content, while S3 stores the originals.
- Heroku[2] is another service that uses AWS as its provider, but offers more user-friendly features and built-in functionality.

[1] https://aws.amazon.com/ec2/pricing/
[2] https://heroku.com