

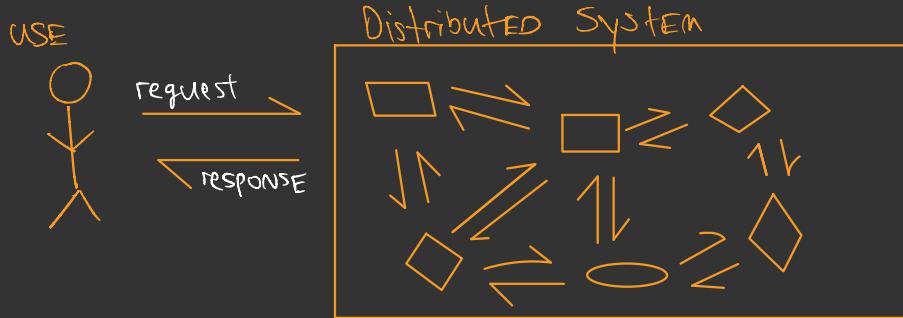
SYSTEM DESIGN

Ivan S
2022



DISTRIBUTED SYSTEMS

- Always keep it simple
- A lot to manage, use when NEED to only
- group of computers working together, complexity hidden from users, feel like interacting w/ single comp.



FALLACIES OF DISTRIBUTED SYSTEMS

Network is reliable

Topology doesn't change

- Lots of things can go wrong, need to handle them somehow

Latency is Zero

network is secure

- Client can't find server
- Server crash mid request
- Server response is lost
- Client crashes

Bandwidth is infinite

Only one administrator

Transport cost is zero

BENEFITS

CHARACTERISTICS

- No shared clock
- No shared memory
- Shared resources
- Concurrency and Consistency

leads to clock drift

COMMUNICATION

- Different parts of Distributed System need to be able to talk
- Requires agreed upon format or protocol

- More reliable, fault tolerant
- Scalability
- Lower latency, increased performance
- Cost effective

PERFORMANCE METRICS

Scalability

- Ability of a system to grow and manage increased traffic
- Increased volume of data or requests

Achieve growth w/out risk to performance

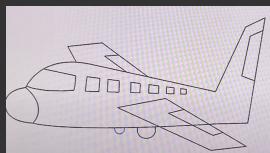
Availability

- Amount of time a system is operational during a period of time
- Poorly designed software requiring downtime for updates is less available

most important for end-users

Reliability vs Availability

- Reliable system is always an available system
- Availability can be maintained by redundancy, but system may not be reliable
- Reliable software will be more profitable because providing same service requires less backup resources
- Requirements will depend on function of the software



PLANE - once in the air, MUST be 100% reliable; tradeoff will always depend on use-case

Reliability

- Probability a system will fail during a period of time
- Slightly harder to define than hardware reliability

Keep working even if hardware or software fails > automated testing, tooling to predict & compensate hardware failures

$$\begin{aligned} \text{Avail \%} &= (\text{available time} / \text{total time}) \times 100 \\ &= 23 \text{ hr} / 24 \text{ hr} \times 100 \\ &= 95.83\% \end{aligned}$$

Measure w/ MTBF

$$\frac{\text{Total elapsed time} - \text{total downtime}}{\# \text{ of failures}}$$

$$24 \text{ hr} - 4 \text{ hr} / 4 \text{ failures} = 5 \text{ hr MTBF}$$

Availability	Annual Downtime
99%	3 days, 15 hours, 40 minutes
99.9%	8 hours, 46 minutes
99.99%	52 minutes, 36 seconds
99.999%	5.26 minutes

common avail metric SLO

Efficiency

- How well the system performs
- Latency and throughput often used as metrics

Manageability

- Speed and difficulty involved with maintaining system
- Observability, how hard to track bugs
- Difficulty of deploying updates
- Want to abstract away infrastructure so product engineers don't have to worry about it

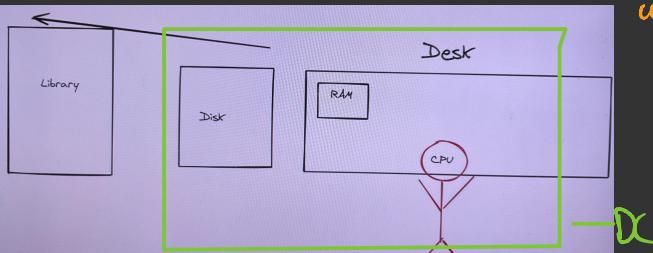
Numbers Programmers should know

Scaled to seconds
to make relatable

LATENCY

Actual Time

Access Type	Time	Converted Time
CPU Cycle	.3 ns	1 S
CPU L1 Cache	1 ns	3 S
CPU L2 Cache	3 ns	9 S
CPU L3 Cache	13 ns	43 S
Main Memory(RAM)	120 ns	6 minutes
SSD	150 micro seconds	6 days
HDD	10 ms	12 months
SF to NYC	40 ms	4 years
SF to Australia	183 ms	19 years



LATENCY TAKEAWAYS

- Avoid Network calls whenever possible
- Replicate data across data centers for disaster recovery as well as performance
- Use CDNs to reduce latency
- Keep frequently accessed data in memory if possible rather than seeking from disk, caching

TIME CONVERSIONS

$$60 \text{ Seconds} \times 60 \text{ Minutes} = 3600 \text{ Seconds per Hour}$$

$$3600 \times 24 \text{ hours} = 86400 \text{ Seconds per Day}$$

$$86400 \times 30 \text{ days} = 2,592,000 \text{ seconds per Month}$$

- CPU = brain, info off the top of your head
- RAM = book on your desk, quick reference
- Disk = bookshelf in house, walk to get
- Network call outside datacenter = travel to a public library to get info

COMMON DATA TYPES

- Char - 1 byte
- Integer - 4 bytes
- UNIX Timestamp - 4 bytes

Data CONVERSATIONS

8 bits	→ 1 byte
1024 bytes	→ 1 Kilobyte
1024 Kilobytes	→ 1 Megabyte
1024 Megabytes	→ 1 Gigabyte
1024 Gigabytes	→ 1 Terabyte

Instagram Example

- Estimate total number of requests app will receive
- Average Daily Active Users X average reads/writers per user

TRAFFIC ESTIMATES

$$10 \text{ Million DAU} \times 30 \text{ photos viewed} = 300 \text{ Million Photo Requests}$$

$$10 \text{ Million DAU} \times 1 \text{ photo upload} = 10 \text{ Million Photo Writes}$$

$$300 \text{ Million Requests} // 86400 = 3472 \text{ Requests per Second}$$

$$10 \text{ Million Writes} // 86400 = 115 \text{ Writes per Second}$$

- Writes per day x Size of write x Time to store data

$$10 \text{ Million Writes} \times 15MB = 15 \text{ TB per day}$$

$$15 \text{ TB} \times 365 \text{ days} \times 10 \text{ years} = 55 \text{ Petabytes}$$

Storage

MEMORY

- Read Requests per day x Average Request size x .2

$$300 \text{ Million Requests} \times 500 \text{ Bytes} = 150 \text{ Gigabytes}$$

$$150GB \times .2(20\%) = 30 \text{ Gigabytes}$$

$$30GB \times 3(\text{replication}) = 90 \text{ Gigabytes}$$

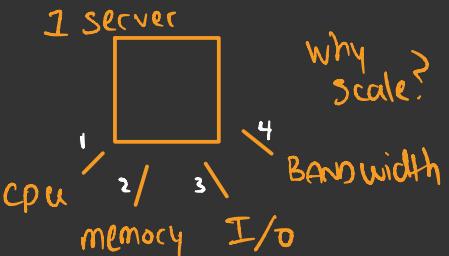
BANDWIDTH

- Requests per day x Request size

$$300 \text{ Million Requests} \times 1.5MB = 450,000 \text{ Gigabytes}$$

$$450,000GB // 86400 \text{ seconds} = 5.2GB \text{ per second}$$

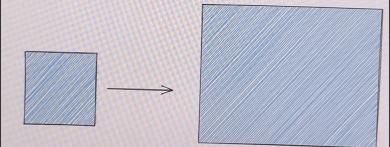
Horizontal vs Vertical Scaling



1. require more processing power
2. need larger amounts of data stored in RAM
3. faster read from storage
4. need bigger amount of data pushed through network; streaming

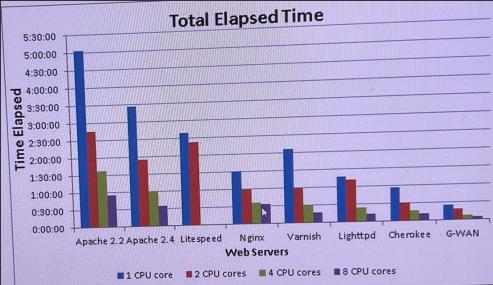
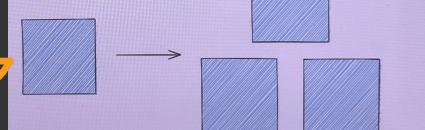
Vertical Scaling

- Easiest Way to Scale an Application
- Diminishing returns, limits to scalability
- Single point of failure

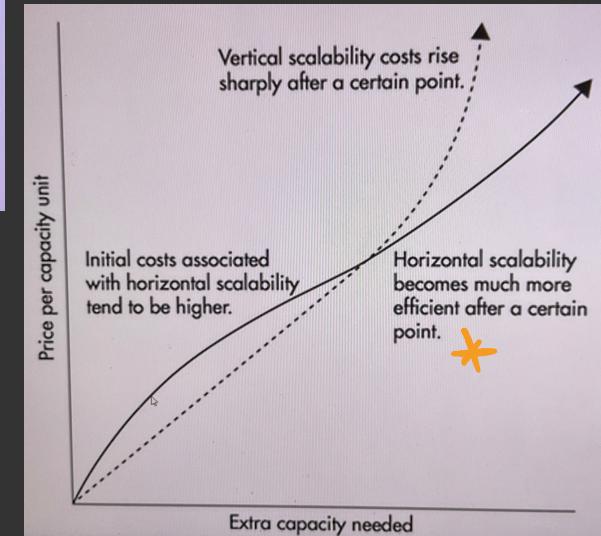


Horizontal Scaling

- More complexity up front, but more efficient long term
- Redundancy built in
- Need load balancer to distribute traffic
- Cloud providers make this easier



Summary Chart

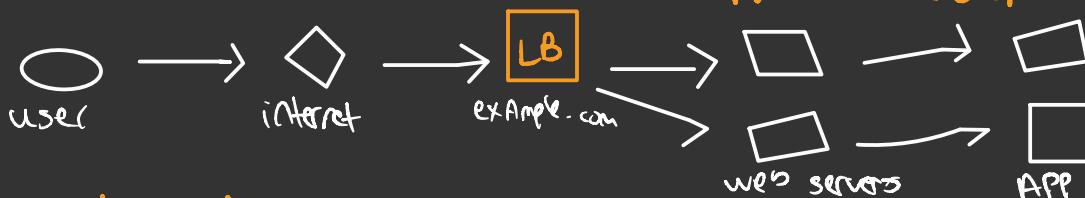


Latency, use Horizontal to improve & prevent S.P.o.F

LOAD Balancers

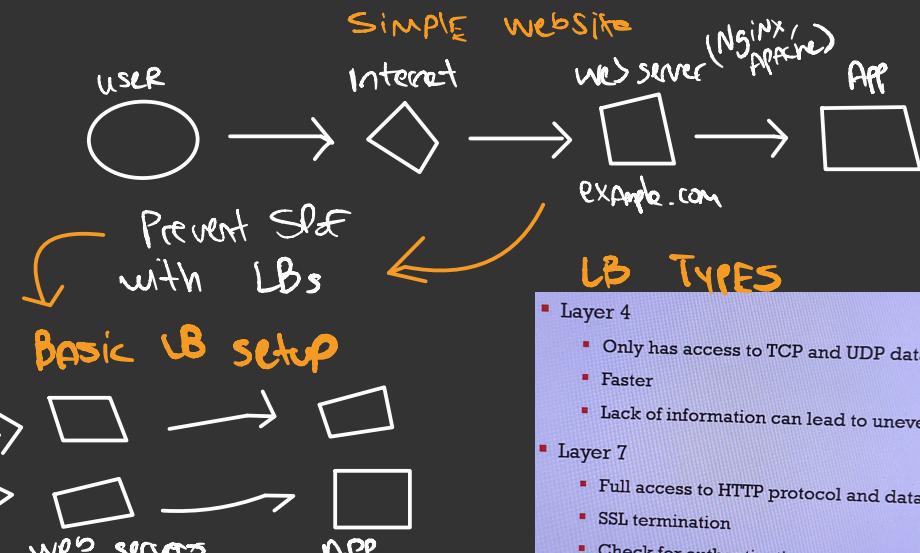
- Balance incoming traffic to multiple servers
- Software or Hardware based
- Used to improve reliability and scalability of application
- Nginx, HAProxy, F5, Citrix

Software Hardware (may lead to vendor lock-in)



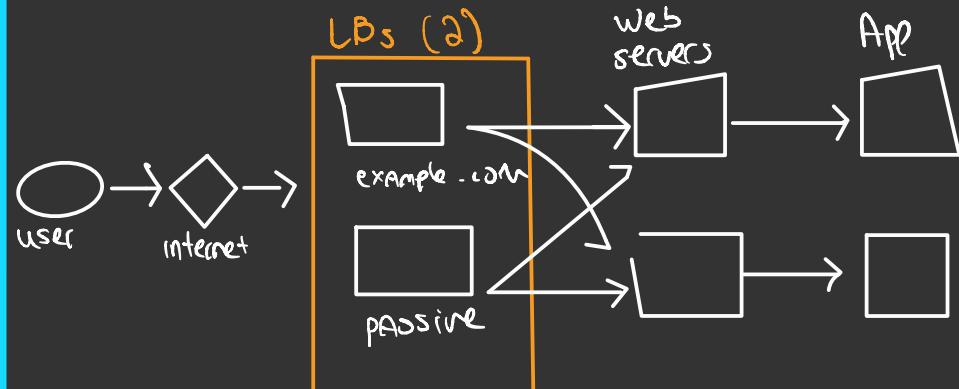
LB Routing Methods

- Round Robin
 - Simplest type of routing
 - Can result in uneven traffic
- Least Connections
 - Routes based on number of client connections to server
 - Useful for chat or streaming applications
- Least Response Time
 - Routes based on how quickly servers respond
- IP Hash
 - Routes client to server based on IP
 - Useful for stateful sessions



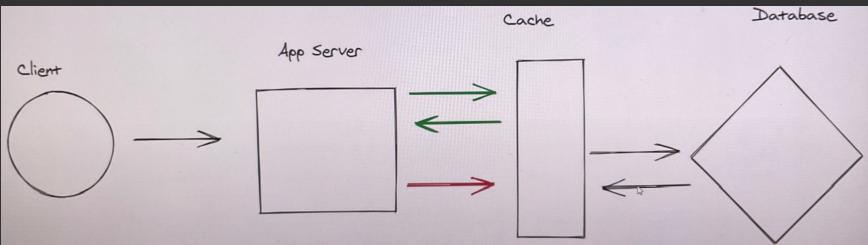
- ## LB TYPES
- Layer 4
 - Only has access to TCP and UDP data
 - Faster
 - Lack of information can lead to uneven traffic
 - Layer 7
 - Full access to HTTP protocol and data
 - SSL termination
 - Check for authentication
 - Smarter routing options

Prox LB SETUP (redundant)



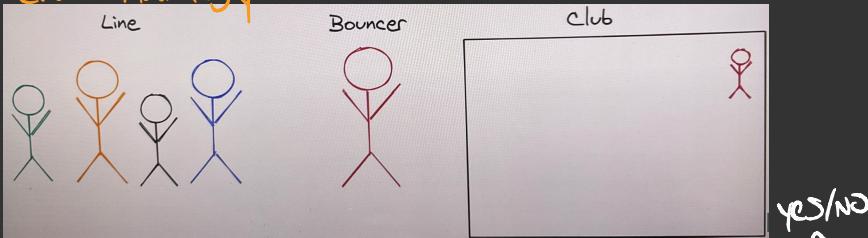
CACHING: TAKE data stored on hard drive/disk and place it in memory

- Improve performance of application
- Save money



- request quicker since retrieving from cache, efficient
- request slower if pulling from DB

CLUB ANALOGY:



- Bouncer has password (in memory)
- Line requests entrance to club, Bouncer verifies pass
- ✗ Bouncer goes inside club each time to ask DB for password → idiot, would piss off line
- Bouncer = cache, Boss = DB, club = App, Line = clients

* CACHE is usually a giant in memory HASH TABLE

- Works same as traditional cache
- Has built-in functionality to replicate data, shard data across servers, and locate proper server for each key

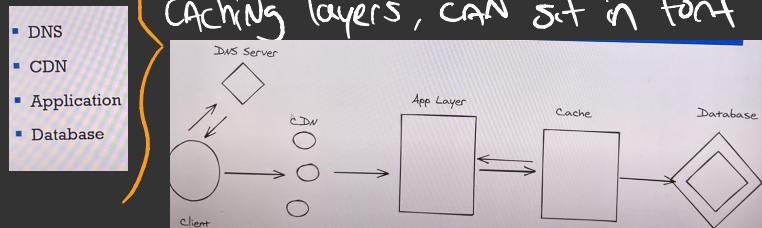
DISTRIBUTED
CACHE
ACTIVE, PASSIVE

Speed
&

Performance

- Reading from memory is much faster than disk, 50-200x faster
- Can serve the same amount of traffic with fewer resources
- Pre-calculate and cache data
- Most apps have far more reads than writes, perfect for caching

CACHING layers, can sit in front of



CACHE Eviction

- Needed for 2 reasons:
- Preventing stale data
- Caching only most valuable data to save resources

TTL

- Set a time period before a cache entry is deleted
- Used to prevent stale data

STRATEGIES

- Least Recently used
- Once cache is full, remove last accessed key and add new key
- Least Frequently used
- Track number of times key is accessed
- Drop least used when cache is full

- How to maintain consistency between database and cache efficiently
- Importance depends on use case

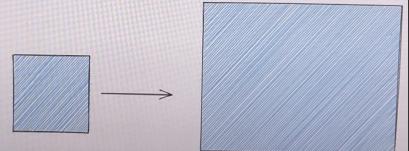
CACHE
CONSISTENCY

DATABASE Scaling

- Most web apps are majority reads, around 95%+

Vertical Scaling

- Get a bigger server
- Easiest solution when starting out



INDEX

- Index based on column
- Speeds up read performance
- Writes and updates become slightly slower
- More storage required for index

Denormalization

- Add redundant data to tables to reduce joins
- Boosts read performance
- Slows down writes
- Risk inconsistent data across tables
- Code is harder to write

- Indexes
- Denormalization
- Connection pooling
- Caching
- Vertical scaling

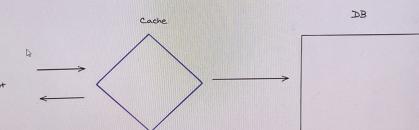
Scaling Techniques

Connection Pooling

- Allow multiple application threads to use same DB connection
- Saves on overhead of independent DB connections

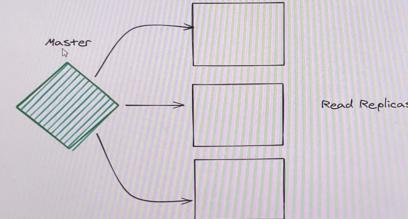
Caching → BEST WAY

- Not directly related to DB
- Cache sits in front of DB to handle serving content
- Can't cache everything



Replication & Partitioning

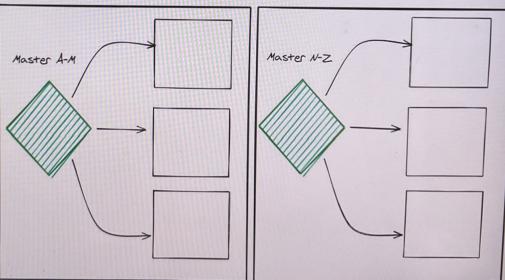
- Create replica servers to handle reads
- Master server dedicated only to writes
- Have to handle making sure new data reaches replicas



READ
Replicas

Horizontal partitioning

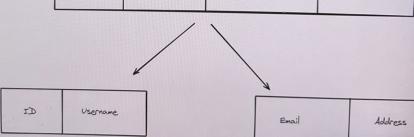
- Schema of table stays the same, but split across multiple DBs
- Downsides – Hot Keys, no joins across shards



Partitioning,
Sharding

- Divide schema of database into separate tables
- Generally divide by functionality
- Best when most data in row isn't need for most queries

ID	Username	Email	Address
----	----------	-------	---------



vertical
partition
* choose NOSQL DB to scale , you
upfront know the tradeoffs

DESIGN YouTube

1. Gather Reqs:

- Video upload
- View uploaded videos
- Search for videos
- Track stats of videos
- Comment on videos

2. Capacity Estimates:

• Storage, bandwidth

3. Key Data:

500 Hours of Video uploaded per Minute

1 Billion Hours of Video watched per day

Storage Estimates:

500 hours of video uploaded per minute

$500 \times 60 = 30,000$ minutes of video per minute

$30,000 \times 5\text{GB} // 1,000 = 150\text{GB}$ uploaded per minute

$150\text{GB} \times 60 \times 24 // 1,000 = 2.16\text{TB}$ per day

2.16PB per day

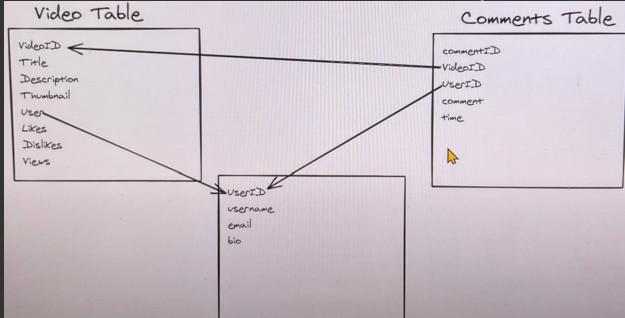
Bandwidth Estimates:

1 Billion hours of video watched daily on YouTube

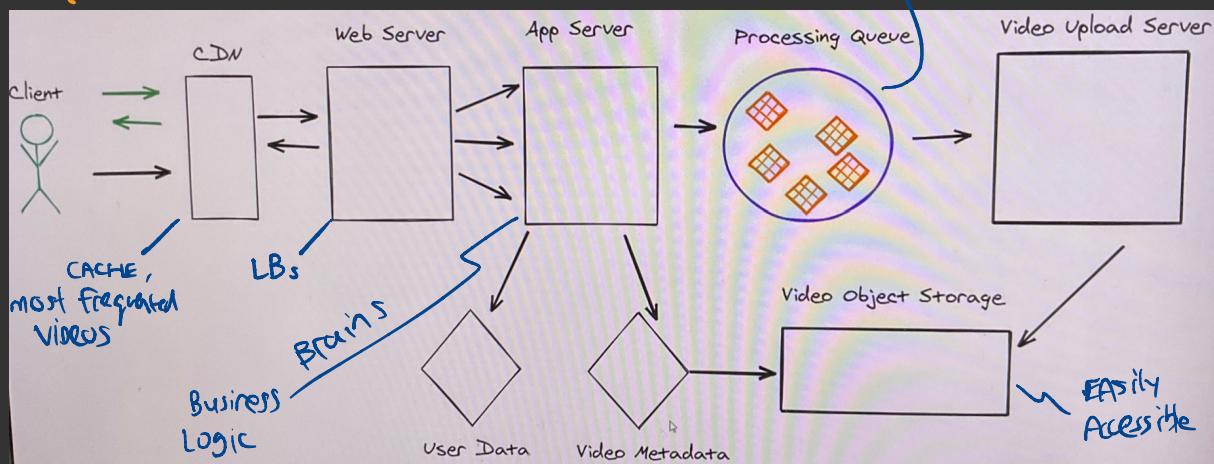
$1,000,000,000 \text{ hours} \times 3\text{GB per hour} // 1000 = 3,000,000 \text{ TB}$

3,000 PB bandwidth/day

DB Design: YouTube user custom RDS setup with high scaling SQL



System Architecture



- Define Requirements
- Make estimates
- High level design
- Specific components

Summary

SYSTEM DESIGN 101

